DOCTOR OF PHILOSOPHY

Solver Tuning with Genetic Algorithms

Xu, Hu

Award date:
2015

SOLVER TUNING WITH GENETIC ALGORITHMS

Hu Xu

A thesis submitted for the degree of Doctor of Philosophy

December 2015

To my family

Acknowledgements

I would like to express my sincere thanks to my supervisor Dr Karen Petrie for providing me with the grateful effort, advice and enthusiasm to successfully complete this thesis. She is not only a unique supervisor in my PhD studying but also a mentor for my research life. She provided vast creative research space and self-improvement opportunities for me. Her support in helping me in the development of the thesis is much appreciated.

I would like to thank my wife Li Xuexin for supporting me throughout this. Thanks for her supporting whatever difficulty I met. I would also like to thank my parents: Xu Xiaojiang and Wang Jinfeng. Thanks for their encouragement during those years. I would thank them for supporting me whatever difficulty I met.

I would like to thanks those people that give me different support during my PhD studying. The list that follows is by no means exhaustive and in no particular order: Keith Edwards, Jianguo Zhang, Iain Murray, Alison Pease, Iain Martin, Wei Xu, Janet Hughes, Anne Millar, Ann enny, Sharon Sturrock, Chris Reed, Emanuele Trucco, Ekaterina Komendantskaya, Derek Brankin, Mahamadou Niakat, Andy Cobley, Mark Snaith, Chris Jefferson, Neil Moore.

Declaration

Candidate's Declaration

I, Hu Xu, hereby declare that I am the author of this thesis; that I have consulted all references cited; that I have done all the work recorded by this thesis; and that it has not been previously accepted for a degree.

Supervisor's Declaration

I, Karen Petrie, hereby declare that I am the supervisor of the candidate, and that the conditions of the relevant Ordinance and Regulations have been fulfilled.

Associated Publications

Parts of the thesis have appeared in the following publications which have been subject to peer review:

1. Hu Xu, Karen E. Petrie, Iain R. Murray. Using Self-learning and Automatic Tuning to Improve the Performance of Sexual Genetic Algorithms for Constraint Satisfaction Problems. ICCSW2013:page 128-135

2. Hu Xu, Karen Petrie, Self-Learning Genetic Algorithm for Constraint Satisfactions Problem. ICCSW 2012: page 156-162

3. Hu Xu and Karen Petrie. A Sexual Genetic Algorithm Based System For Automatically Tunning Constraint Satisfaction Problems. ARW 2013

4. Hu Xu, Karen Petrie and Keith Edwards, Genetic Based Automatic Configurator for Minion. The 17th International conference on Principles and Practice of Constraint Programming Doctoral Program 2011

Other peer reviewed publications during PhD:

Wei Xu, Jianguo Zhang, Hu Xu, Maojun Zhang, Height Estimation of Urban Buildings Using Angle Consistency of Borderlines of Roofs, ICME 2013.

Abstract

Currently the parameters in a constraint solver are often selected by hand by experts in the field; these parameters might include the level of preprocessing to be used, the variable ordering heuristic or the suitable modelling approach. The efficient and automatic mechanism of parameters tuning for a constraint solver is a step towards making constraint programming a more widely accessible technology. Two types of tuning algorithms are discussed in this thesis: single instance tuning algorithms and instance-based tuning algorithms. A standard genetic based algorithm and a sexual genetic based algorithm are proposed and implemented to deal with the single instance tuning. As an instance-based tuning algorithm, the self-learning genetic algorithm, which suggests or predicts a suitable solver configuration for test instances by learning from train instances, is proposed in this thesis. To improve the efficiency of the instance-based tuning in further, a self-learning sexual genetic algorithm, which combines the self-learning mechanism with the sexual genetic algorithm, was discussed. The experiments in the thesis demonstrate how genetic algorithms are implemented and adapted to aid in parameter selection for constraint solvers. Genetic algorithms were implemented as the fundamental algorithm for tuning and the parameter sensitivity of genetic algorithms is also discussed in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Problems often consist of choices. Making an optimal choice which is compatible with all other choices is difficult. Constraint programming (CP) [75] is a branch of Artificial Intelligence [80], in which computers help us to make these choices.

Constraint programming is a multidisciplinary technology combining computer science, mathematics, and operations research [31] which is a discipline that deals with the application of advanced analytical methods to help make better decisions. Constraints arise in design, configuration, planning, scheduling, diagnosis, testing, and in many other contexts. Constraint programming can solve problems in telecommunication, e-commerce, electronics, bioinformatics, transportation, network management, supply-chain management, and many other fields.

In constraint programming, different constraint propagations [17] or search strategies are implemented in order to find the solution(s) of the problems. The search strategies and propagation mechanism chosen in a given problem lead to different search time for the solution(s). Therefore, the selection and utilisation of suitable constraint propagations and search algorithms for a given constraint problem is an important part of constraint programming.

A constraint solver is a platform that a researcher could utilise to find the results of a constraints satisfaction problem by picking a suite of which are often available constraint propagations and search strategies. Often there is a parameter variable available to choose a propagation or search strategy in the constraint solver. Efficiently tuning a constraint solver, which finds a suitable parameters setting, will shorten the search time and reduce the running cost to find a solution. Thus tuning the solver's parameters [66] is one significant step of increasing the search speed for a constraint solver.

Currently, the job of tuning the parameters is done manually; a skilled user selects the most suitable preprocessing method by using previous experience from similar classes of problems. In most cases, the best constraint propagations and search strategies used in similar classes of problems will provide a useful clue to aid the user's selection. However, this learning curve could be a barrier to novice users in learning how to efficiently use a CP solver [69]. The user must learn about what each parameter in a constraint solver does before trying the numerous possible parameter settings. Hutter et. al [55] showed that manual tuning often leads to highly inferior performance. In other words manual tuning, in practice, wastes the user's time and needs a good understanding of the tuning objects. This is especially a issue for the beginner in CP.

In [70] it pointed out the ultimate goal of automatic search is to build systems that were able to autonomously solve problems, and to acquire and even discover new knowledge, which could be reused later. Therefore a feasible automatic mechanism for a CP solver is worthy of further study.

Genetic algorithms (GAs) are classic global optimisation methods posed by John Holland [51], which mimic the competition of organisms in nature and the mechanisms of evolution. GAs are usually implemented in a computer simulation in which a population of abstract representations of candidate solutions to an optimisation problem evolves towards better solutions. GAs are widely applied to optimisation problems such as configuration problems. [5] has proposed a gender-based genetic algorithm for the automatic configuration of algorithms; it showed that the genetic-based approach is feasible for automatic

configuration.

Therefore the idea of combining GAs and constraint programming seems worth exploring further and it is expected that automatic tuning will lead to improvements over manual tuning by users. Some strategies, such as ParamILS [56], have demonstrated the practicality and efficiency of automatic configuration for constraint solvers. However, the general framework of combining GAs with constraint programming has not been achieved. In light of this situation, some new genetic based approaches such as a sexual genetic algorithm, a self-learning genetic algorithm, and a self-learning sexual genetic algorithm were proposed to help tuning constraint solver automatically in this thesis. Meanwhile the parameter sensitivity of genetic algorithms itself in solving those tuning problems has not been explored, and it will be discussed in this thesis.

1.2 The Existing Automatic Search Approaches

In [48], it was mentioned that the automatic search, which finds the most suitable and correct setting for solving a given problem, caught many researchers' attention in Artificial Intelligence (AI) and was investigated for many years as the selection problem [90] due to its importance. The field of automatic search, which has experienced a renaissance in the past decade, tries to overcome these limitations of manual tuning.

According to the type of the optimisation problems and the tuning parameters, the existing tuning algorithms or search algorithms which can be applied in tuning could be classified into four groups by the research focus. The first group is utilised for tuning such parameters which have continuous domain. Based on the concept of self-adaptation of the classic evolution strategy, [49] posed a Covariance Matrix Adaptation Evolution Strategy which adapts arbitrary normal mutation distributions. The results show that the Covariance Matrix Adaptation Evolution Strategy is feasible for difficult non-linear, non-convex, black-box optimisation problems in a continuous domain. The internal parameters of Covariance Matrix Adaptation Evolution Strategy could self-adapting efficiently.

In the constraint optimisation, mesh adaptive direct search algorithms [9], which are based on the generalised pattern search [8], were posed by Audet. In mesh adaptive direct search algorithms, a local exploration, called polling strategy, was utilised in an asymptotically dense set of directions in the space of optimisation variables. From the description of those approaches, it shows that those approaches have a great advantage on dealing those parameters with the continuous domain. But the range of the tuning parameters was limited to the continuous domain.

The classification tuning or searching, which is based on categorical parameters, is the second trend in the tuning algorithms. Composer [43] is one of the classification searching systems for specific distributions of problems and constraints. In the classification tuning or searching strategies, they firstly find some set of configurations or parameter settings by statistical or other evaluation methods. The better or best parameter settings will be picked up from the chosen parameter settings. Composer is an example to illustrate the classification tuning. Composer implements a hill-climbing search through a space of the possible domain to find the right solution.

Based on the knowledge of machine learning, Birattari [18] posed an automatic configuration mechanism for algorithm parameters, called F-Race algorithms. The main idea of the F-Race is a statistical method by selecting the best configuration from F, a set of candidate configurations, which were gained by stochastic evaluations.

CALIBRA [2] and ParamILS have also shown the efficiency of automatic configuration. CALIBRA implements Taguchis fractional factorial experimental designs [93] coupled with a local search procedure to find a better configuration for the optimisation problem. The aim of CALIBRA is not to find the best configuration but the better configuration in requested time. The mechanism of ParamILS is to implement some local search algorithms with adaptive capping technique for automatic configuration. ParamILS implements a special iteration technique that gathers statistics on which parameters are important. The current best parameter set would be replaced only if a new parameter set has been evaluated on at least as many training instances as the current best. However those mechanisms are

only designed for the categorical parameters as mentioned at beginning.

The third research direction in tuning algorithms is a generic configuration for general parameters. One of the most successful attempts is what Ansótegui [5] has posed: a gender-based genetic algorithm (GGA) for the automatic configuration of algorithms. He uses a variable tree (AND/OR Search Trees) to divide variables into several parts which can be optimised independently. Based on all the ideas of these tuning algorithms, this thesis will explore the possibility which would combine the machine learning and non-model-based configuration approach to tuning the constraints solver.

The fourth trend in developing tuning is model-based or case-based parameter tuning. Sequential Parameter Optimization [12], which combines classical and modern statistical techniques is a typical model-based optimisation. Sequential Parameter Optimization determinates the optimal configuration under the experimental analysis of a few of the design points. CP-Hydra [82] implements case-based reasoning to suggest the configuration for unseen instance by exploiting the experience for the existing instance. Sequential Parameter Optimization and CP-hydra shows that the main idea of the model-based or instance-based parameter tuning is to suggest the parameter setting by analysing the existing instance.

[66] posed a machine learning based strategy called ensemble classification for automatic tuning. The ensemble classification is an approach which combines several machine learning algorithms such as decision trees and neural networks. According to the type of the optimisation problems, the ensemble classification will pick up the right strategy.

Most recently, a Sequential Model-based Algorithm Configuration (SMAC) [54] was introduced in 2010. This approach proposes to generate a model over the solver's parameters to predict the likely performance. This model can be anything from a random forest to marginal predictors. This model is used to identify aspects of the parameter space, like what parameters are the most important. Possible configurations are then generated according to this model and compete against the current incumbent. The best configuration continues onto the next iteration. While this approach has been shown to work on some problems, it ultimately depends on the accuracy of the model used to capture

the interrelations of the parameters.

Based on the gender genetic algorithm, [64] proposed an algorithm called the instance-specific algorithm configuration (ISAC). ISAC is based on the integration of the configuration algorithm GGA and the recently proposed stochastic off-line programming paradigm. ISAC will firstly normalise the corresponding feature vectors of a list of training instances. Then the g-means algorithm will be implemented to cluster the training instances based on the normalised feature vectors. Next, for each cluster of instances ISAC computes favourable parameters using the instance-oblivious tuning algorithm GGA. For future test instances, its feature vectors will be normalized. ISAC will find a cluster such that the normalised feature vector of the training instance is close enough to the cluster centre. Then ISAC will suggest the parameters of the cluster for the training instance.

It shows that a variety of strategies were implemented to those attempts on automatic tuning. Although there are many successful attempts at implementing genetic algorithms for automatic tuning, there is no general framework for implementing genetic algorithms to tuning a constraint solver, as mentioned above.

1.3 The Aim of the Thesis

From the literature review in the above section, it shows the aim of the thesis is to proposed genetic based algorithms to tune a constraint solver in different situation. Therefore there are three main research questions in the thesis that need to be answered. The first one: How may a genetic-based algorithm be created for automatically tuning a constraint solver? The second one: What kind of strategies may be implemented to improve those genetic-based tuning algorithms? The last one: How will those new genetic-based methods be evaluated for their performances?

Design A Genetic Based Automatic Tuning Mechanism For Constraint Solvers

A feasible automatic tuning mechanism is the key and the first step for tuning constraint solvers. Therefore the investigation of the basic concepts of the genetic algorithm and constraint programming is required. A genetic based tuning mechanism will be designed to connect the genetic algorithms with a constraint solver for tuning. The thesis will justify the genetic based tuning mechanism.

Explore The Possible Hybrid Tuning Approaches For Constraint Solvers

To deal with different tuning situations with a higher efficiency, some new tuning approaches, which combine genetic algorithms with other optimisation strategies, will be explored. In this thesis those hybrid approaches were proposed for two kinds of tuning situation: the single instance tuning which tunes a constraint solver for one instance; and the instance-based tuning which consists training instances and testing instanced.

[40] mentioned that the quality of parameter setting of the genetic algorithm will greatly affect the result of the search speed and convergence. Therefore the parameter sensitivity of the genetic algorithm itself in those hybrid tuning approaches will be discussed.

Evaluate The New Genetic Based Hybrid Tuning Approaches

Evaluating those new genetic based hybrid tuning approaches is a vital part in the thesis. To justify the efficiency of those new tuning approaches, their performance will be compared with some recent existing algorithms as mentioned in the literature review such as GGA ParamILS and SMAC.

1.4 The Structure of the Thesis

This thesis will use six chapters to answer the three main research questions.

Chapter 2 will investigate the operators in the standard genetic algorithm, such as the selection, mutation, and crossover. After the experiments, it will state the basic features

and the performance.

Chapter 3 will introduce various definitions in constraint programming, different constraint propagation strategies, and search algorithms. Minion [37], which is one of the most efficient constraint solvers in the world, will be presented and discussed in this chapter. Four classic constraint satisfaction problems will be introduced.

Chapter 4 will illustrate how the standard genetic algorithm works to help tune the parameters in the Minion. A genetic based algorithm will be discussed and implemented by comparing with random selection strategy.

The sexual genetic algorithm and the elitism selection will be explained in Chapter 5. The efficiency of this SGA will be tested by comparison with a standard GA for the Travelling Salesman Problem. The performance of SGA will compare with the gender genetic algorithm.

The self-learning genetic algorithm (SLGA) will then be introduced and applied to select a suitable parameter set as an instance-based tuning algorithm in Chapter 6. It will demonstrate how SLGA gains clues from the small training instances for the large testing instances. The performance of the SLGA will also be verified by comparing with ParamILS .

Finally, another instance-based tuning algorithm the self-learning sexual genetic algorithm (SLSGA) will be analysed by comparing the sexual genetic algorithm in Chapter 7. Meanwhile the new approach will compare with SMAC. The thesis will conclude by making some conclusions and presenting the future work.

Chapter 2

Standard Genetic Algorithm

This chapter will explore and consider the operators and the basic performance of the traditional standard genetic algorithm based on [40]. The principle and strategy of the standard genetic algorithm operators i.e. selection [67, 110], mutation [79] and crossover [102] will be investigated. In order to understand the operators and evolutionary rule of genetic algorithm, the basic feature of standard genetic algorithm will be discussed. Therefore we will firstly present the operators, the optimisation mechanism of the standard genetic algorithm in section 2.1. The influence of those parameters in genetic algorithms such as crossover and mutation will be illustrated and discussed in a benchmark function. Next, some experiments will be implemented to monitor the performance of the standard genetic algorithm after presenting the fundamental features of the standard genetic algorithm. Meanwhile the influence of the starting population is also explored. Finally we will draw some conclusions for this chapter in the last section.

2.1 Introduction

Genetic Algorithms (GAs), which mimic natural evolution, have been applied as search based algorithms based on Darwin's theory of evolution [22]. GAs mainly imitate the recombination of chromosomes which are also sometimes called individuals. In fact each chromosome in GAs presents a possible solution to the optimisation problem. The

chromosome in GAs is often represented as a simple binary string, an integer array or other data structures. In GAs some chromosomes constitute a biological population. In GAs a population is a set of possible solutions to the optimisation problem. GAs usually create a random chromosomes group which is called the old population at beginning. To gain the chance that acquires perfect solutions (referred as chromosomes in Genetic algorithms); the new population is created by mating of chromosomes in an old population. Here the new population is called the offspring and old population is referred to as the parents in order to presents the relationship between the new population and the old population vividly. Since the offspring is generated from parents, the offspring has many similar features of its parents. The whole or parts of the chromosomes in parents are carried to the offspring. Those chromosomes store the information of the parents.

The early genetic algorithm is called standard genetic algorithm (SGA), which just includes a basic selection policy, a single point crossover and a single point mutation. Although the selection policy, crossover and mutation were always adapted and revised to various variants for different optimisation problems, the evolutionary mechanism of the genetic algorithms is commonly composed of the following several parts:

Encoding

Fitness Function

Selection

Crossover

Mutation

In order to create a preferable new generation, a specific function which is called the fitness function judges the quality of each chromosome in the population. Then according to the fitness of each chromosome, the selection strategy will help optimal individuals gaining more chances to be picked up into the mating pool to create ideal offspring. Such selection mechanisms promise that the higher fitness individuals in the old population have more opportunities than the poor fitness individuals to be selected to generate the next generation.The mutation and crossover which are the classic operators in GAs provide

Colour	chestnut	black	gray	white
Binary encoding	00	01	10	11

Table 2.1: GA Binary Encoding of a horse Colour

GAs the ability of changing the chromosome in the new generation.

Each part is important and indispensable for the genetic algorithms optimisation. In order to understand the mechanism and the principle of the genetic algorithms, each part will be presented in more details in the following section.

2.1.1 Encoding

Definition 2.1.1 *Encoding The encoding in genetic algorithms is a procedure that transfers each possible solution of the optimisation problem to some kind of string as a chromosome.*

The first step in the genetic algorithms is encoding. From the definition 2.1.1, it shows that the aim of the encoding is to transfer the solutions to Strings. The proper string structure is the foundation to implement the later operators in genetic algorithms. The common encoding in genetic algorithm is a binary encoding [40]. Each chromosome is a string which is composed of 0 and 1. To understand the encoding mechanism, the following will illustrate how to encode a real problem. Here assume that a farmer introduce a group of horses which have different colours, ages and strains. But what will happened if he lets the horses breed freely? Is it possible to get a horse which has a strong ability on breed? The GA can help the farmer realise it.

The farmer is only concerned with three characteristics of the horse: colours, ages and strains. The breed ability of each horse is related with those three characteristics. Therefore each horse can be encoded with a chromosome which has the information of horse's three characteristics. For example the colour of horse is chestnut, black, gray and white. In binary encoding the four colours can be presented by 2 bit binary number see table 2.1.

The age of horse is a range from 1 to 8, and 3 bit binary number can presents all the 8

ages. The strains of horse can also be presented in a similar way. Then one of the horse in the group can be encoded in the following format see table 2.2.

The chromosome after encoding: 1100011

The horse feature: A one years old, white, Arabian horse

And each horse in new generation also can be presented as a chromosome. So the farmer can predict roughly the new generation of horses by observing the evolution of chromosomes. Features of the horse can be added or deleted with the requirement of a real problem. For example the gender or the weight of horse all can be encoded as well if they are part of the problem. Therefore, according to the value range of a research object or the requirement of an optimisation problem, the binary encoding length (or the chromosome length, as it is referred to in GAs) is changed.

2.1.2 Fitness Function

Definition 2.1.2 *Fitness Function*

Fitness function or objective function is a particular type function which evaluates the merit (fitness value) of each chromosome which is a possible solution for an optimization problem.

Fitness describes the ability of a chromosome to reproduce in biology. In the genetic algorithms, the fitness describes the quality of each individual (solutions). In the problem optimisation, the genetic algorithms use fitness function to evaluate each individual and provide the information to aid evolution. The fitness function gives the information to the selection process to pick the suitable parents and push them to a mating pool.

In the last section each horse was encoded as a chromosome with a horse's three characteristics which are related with breeding ability. In the horse problem we want to find the offspring which has strongest breed ability. So the breed of ability of each horse is selected as the fitness in this optimisation problem.

The fitness function is the function which evaluates the difference between the desired

	colour	age	strain
Horse features	white	1	Arabian horse
Binary format	11	000	11

Table 2.2: GA Binary Encoding of a horse

result and the actual result. It means that a fitness function needs to be built to evaluate the breeding ability of each horse. Here, assume that the breeding ability of each horse is the decimal value of the chromosome which includes the horse's three characteristics: colour, age and strain. Then the fitness function in the horse problem is:

$$F(x) = D(Ch(x))$$

where x is a horse, $F(x)$ is the fitness and the breeding ability of the horse x, $Ch(x)$ is the chromosome which includes the horse's three characteristics, $D(Ch(x))$ is the function to calculate the fitness of the chromosome $Ch(x)$ with the horse's characteristics. To easily understand the fitness calculate function $D(Ch(x))$, here simply assume the fitness of horse is the overall decimal value of each horse characteristic binary coding. There is a white (binary coding: 11, 2 bits) one year old (binary coding: 000, 3 bits) Arabian horse (binary coding: 11, 2 bits). The horse after encoding is 1100011 and the chromosome length is 2+3+2=7. Then the fitness of this horse is

$$1 * 2^{(7-1)} + 1 * 2^{(6-1)} + 0 * 2^{(5-1)} + 0 * 2^{(4-1)} + 0 * 2^{(3-1)} + 1 * 2^{(2-1)} + 1 * 2^{(1-1)}$$
$$= 64 + 32 + 0 + 0 + 0 + 2 + 1$$
$$= 99$$

It is usual to set an ideal fitness as an optimal target after building a fitness function for an optimisation problem. Fitness function can evaluate the quality difference between each horse of the existing group and the ideal horse.

2.1.3 Selection

Definition 2.1.3 *Selection*

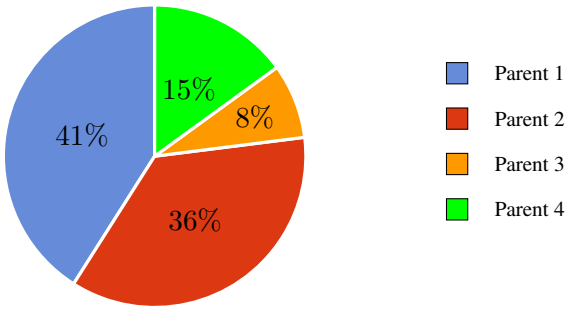


Figure 2.1: Roulette Wheel Selection

Selection in genetic algorithms is a mechanism that helps to select the suitable parent chromosomes for generating the next generation.

To generate more perfect offspring in the new generation, a mechanism is desirable to make the best individuals in parent generation have a better chance in breeding and selection. Selection is such a necessary stage before doing the crossover and mutation in mating pooling. The selection mechanism makes the likelihood of choosing a higher fitness chromosome greater than that of a lower fitness one. Therefore the proportion of good fitness in a mating pool is usually higher than the one in the parent population after the genetic algorithm selection. There are many kinds of selection in the genetic algorithms, but there are two common selections: roulette wheel selection [40] and tournament selection [76].

Roulette Wheel Selection

Roulette wheel selection is a way of choosing individuals from the population of chromosomes in a way that is proportional to their fitness. Roulette wheel does not guarantee that the fittest member goes through to the next generation, merely that it has a very good chance of doing so. Assume there is a wheel. The area size of wheel is the sum fitness of population, and the size of each slice is the fitness of each chromosome. According to the fitness of each chromosome in genetic population, the wheel is divided into a few slices according to the population size. In the population the higher fitness of each individual

means the bigger area of each slice and proportion in wheel. The following is the selection probability formula for each chromosome:

$$P(C_i) = \frac{f_i}{\sum_{i=1}^{n} f_i}$$

Where $P(C_i)$ is the selection probability of each chromosome, C_i is the ith chromosome in population, f_i is the fitness of chromosome C_i and n is the population number.

For example, assume a virtual wheel and there are 4 chromosomes in population. The fitness of each chromosome is the decimal value of each binary encoding chromosome. Figure 2.1 shows the selection proportion of each individual and how selection works.

From Table 2.3 it shows that parent 1 has the probability 41% to be selected and pushed to the mating pool. However parent 4 just has the probability 8% to be selected and pushed to the mating pool. It illustrates that the bigger fitness means the more share of roulette wheel and the higher probability of being selected each individual has. However weaker individuals are not without a chance. And this is good because the mechanism helps to keep the population diversity and avoid falling the local optimal trap. Roulette wheel selection meets the requirement of best survival, and the computation of roulette wheel selection is not huge. Therefore roulette wheel selection is widely applied in genetic algorithm.

	Chromosome	Fitness	Proportion
Parent 1	111000	56	41%
Parent 2	110001	49	36%
Parent 3	001011	11	8%
Parent 4	010101	21	15%
Total		137	100%

Table 2.3: Roulette Wheel Selection

Tournament Selection

Roulette wheel selection is not the only selection in GAs, there is another selection strategy called tournament selection. To compare with the roulette wheel selection, the

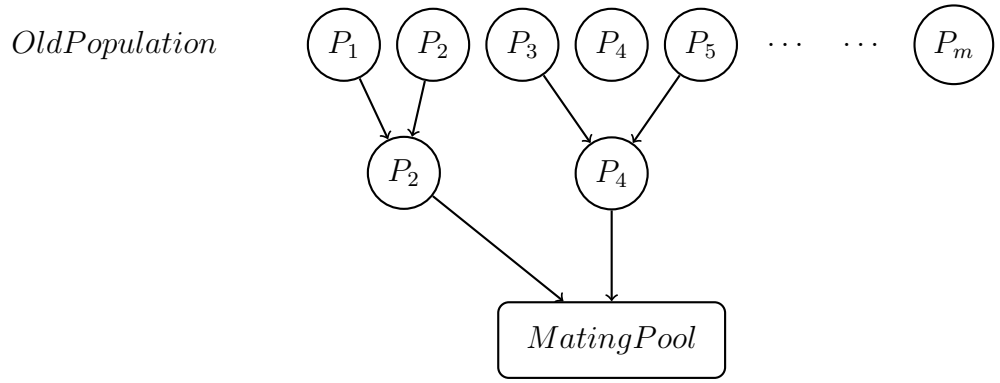


Figure 2.2: Tournament Selection

tournament selection randomly chooses $M(M >= 2)$ chromosomes. Then the highest fitness individual is selected and other individuals are discarded. Each time the best fitness individual is pushed to the mating pool where the individuals will do the genetic recombination to generate new generation until the new population size is the same as the parent population. The tournament selection doesn't have the probability to pick up the smallest fitness parent. In roulette wheel selection each parent has a relative probability to be picked up to the mating pool. This is the big difference between roulette selection and tournament selection. Now we give an example to show the working mechanism of tournament selection. If there are a group of chromosomes which population size is M, tournament selection will choose $N = 2$ individuals each time to decide which has the opportunity to be selected and to push into the mating pool. Tournament selection repeat until the amount of the new individuals in new generation is M. The following figure shows the working mechanism of tournament selection.

Figure 2.2 shows that the old population whose population size is M selects $N = 2$ individuals to compete the position in the mating pool each time. Because the fitness of P_2 is larger than the P_1 one, the P_2 will be picked up to the mating pool on the first competition. By the same way, the fitness of P_5 is larger than P_3 and P_5 will be selected to push into the mating pool. Each time the tournament selection randomly picks up two chromosomes from the old population no matter if they have been selected already or not.

Before Crossover							
Position	1	2	3	4	5	6	7
Parent 1 (Fitness=96)	1	1	0	0	0	0	0
Parent 2 (Fitness=79)	1	0	0	1	1	1	1
After Crossover							
Child 1 (Fitness=111)	1	1	0	1	1	1	1
Child 2 (Fitness=64)	1	0	0	0	0	0	0

Table 2.4: Crossover

All the high fitness individuals which are picked up after tournament selection will do the recombination in mating pool to generate new generation. GAs will repeat the selection and recombination until the new generation size reaches M. The speed of approaching optimal individual of tournament selection is faster than the roulette wheel selection, but tournament selection easily misleads the evolution to the trap of local maximum, because it always keeps the high fitness individuals and loses the variety of population.

2.1.4 Crossover

Definition 2.1.4 *Crossover*

Crossover in genetic algorithms is an operator which randomly swaps the same length strings in each pair of selected chromosomes.

The crossover, which mates parents to produce an offspring, is another very important operator in the genetic algorithms. The crossover attempts to generate an individual which has a higher fitness by swapping parts in the selected chromosomes. Single point crossover is the most classic crossover in genetic algorithm because it can be easily understood and implemented. In single point crossover a crossover operator that randomly selects a crossover point within a chromosome then interchanges the two parent chromosomes at this point to produce two new offspring. Assume there are two parents whose fitness is the decimal value of individuals and the fitness of two parents is 96 and 79. The following is the working way of single point crossover.

According to the chromosome length, which is the length of the encoding bit in a

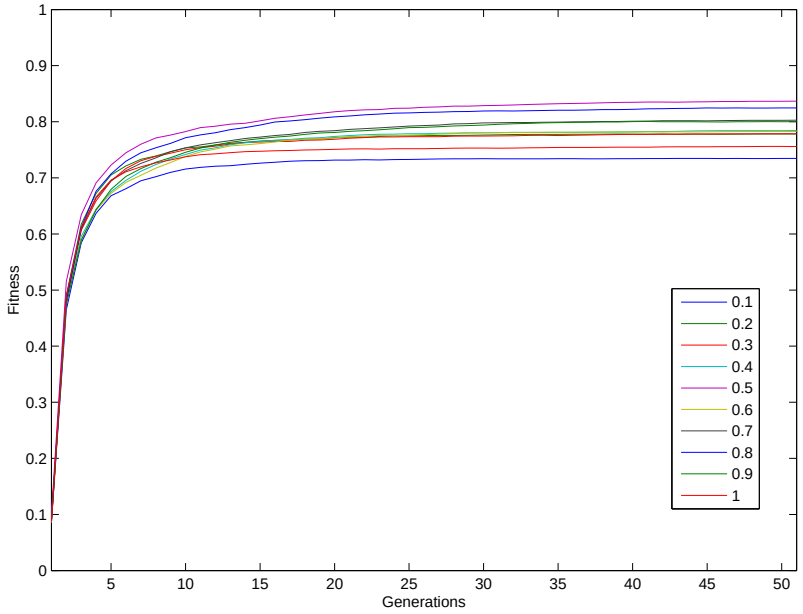


Figure 2.3: Crossover Rate Range Testing

chromosome, a crossover position will be randomly generated between 1 and chromosome length-1 before each crossover. Table 2.4 shows that the crossover splits each parent mentioned above into two at the same position (position=3) in this example and exchanges the same length strings of each parent. The information in parent 1 exchanges the information in parent 2 by this way. As mentioned in the GAs encoding, each chromosome in GAs actually presents a solution to an optimisation problem. Each segment in chromosomes or gene strings, presents partial solutions to the problem. The aim of the GAs is to find the best combination of the superior gene string which could lead the chromosomes to a better or the best fitness. Since the superior gene string could be in any position of any chromosome in the whole population, crossover provides the chance to combine superior gene strings. As table 2.4 shows,the first three bits in parent 1 and the last four bits in parent 2 are superior gene strings. Since the fitness of an individual is the decimal value of each individual, the fitness of the individual can be presented by the sum of the decimal value of the first three and last four bits of an individual. In other words the fitness of the first three bits in parent 1 is greater than parent 2, but the fitness of the last four bits in

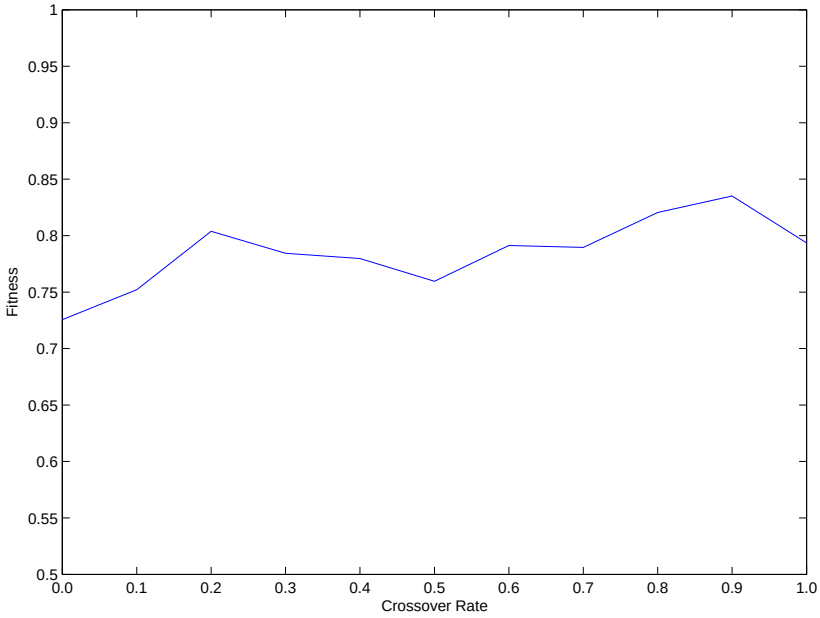


Figure 2.4: The 50th Generation Fitness against Crossover Rate

parent 2 is larger than parent 1. After crossover the child 1 inherits all the superior gene strings from parent 1 and parent 2. Therefore child 1 has the high fitness than its parents after crossover.

Usually before crossover, a parameter called crossover rate C_r which is the probability of crossover happening needs to be set in GA. Crossover rate or crossover probability C_r indicates a ratio of how many couples of chromosomes will be picked for mating. It means that if the crossover rate is 0.6, each two selected parents have the 60% probability to do the crossover in mating. For more understanding on crossover rate and its influence, we choose an example form [40]. In the example the chromosome was decoded as an unsigned 30-bit integer. The fitness function f is the power function:

$$f(x) = (x/c)^n$$

where c has been chosen to normalise x, and n has been chosen as 10. Since the bit string is an unsigned 30-bit integer, c has been chosen as 1073741823.0 $(2^{30} - 1 = 1073741823)$ to normalise x. After normalisation the value of x and $f(x)$ will range from

Parent: 101111 Fitness:47
Child: 111111 Fitness:63

Figure 2.5: Mutation

0 to 1. The optimisation problem seeks the best individual with $f(x)$=1. In the chapter the function was chosen to investigate the influence of crossover rate in GA firstly.

Here assume there are 30 chromosomes in a population and each chromosome's length is 30 bits. The population will do evolution for 50 generations. Crossover rate ranges from 0 to 1 and steps 0.1. 100 different starting populations will be created to do the same evolution. Figure 2.3 illustrates the fitness curves for different values of crossover, plotted against generation number. Each curve represents the average fitness of the population at each generation. Each curve in turn is also the average of 100 different starting populations.

Figure 2.3 and Figure 2.4 demonstrate the influence of the crossover rate in the benchmark function. Figure 2.3 shows that the 50th generation fitness found don't rise with the increase of crossover rate. In order to analyse the result in Figure 2.3 clearly, the 50th generation fitness of each curve presented in drawn graphic in Figure 2.4.

From Figure 2.4 it shows that the crossover rate around 0.9 is peak. And the 50th generation fitness doesn't increase with the raise of crossover rate. The 50th generation fitness actually decreases with the raise of crossover rate when crossover rate is (0.2, 0.5) and (0.9, 1.0). And Figure 2.3 also shows that the proper crossover rate help the SGA to find the optimal result whose value is more than 70% of the best fitness in 50 generations in this benchmark. Although the crossover is done randomly, the mating individuals are selected by comparing the fitness. The child chromosome can inherit the high fitness gene string from parents, but the creation of new gene strings will depend on mutation. The following section will describe the working mechanism of creating new gene string of mutation.

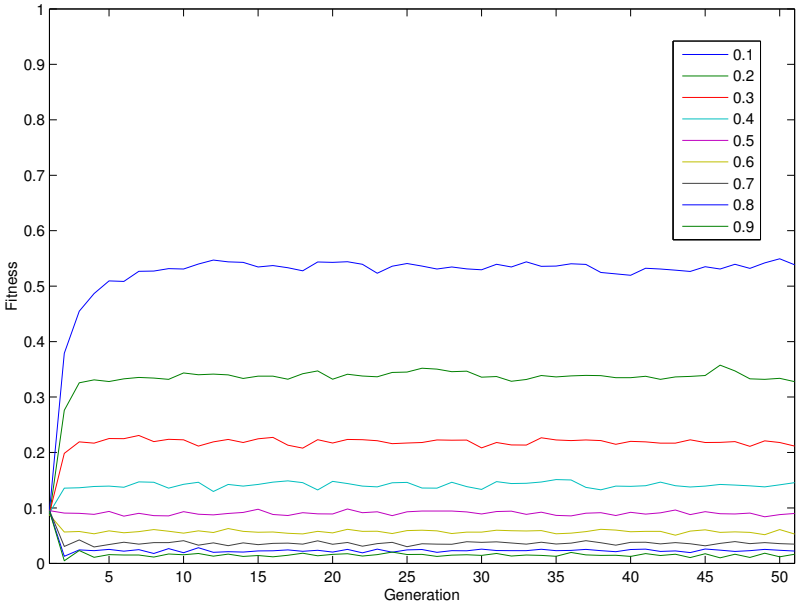


Figure 2.6: Plots of average fitness, versus generation, Mutation rate is [0, 1]

2.1.5 Mutation

Definition 2.1.5 *Mutation*

Mutation alters one or more gene values in a chromosome from its initial state to any other state at random.

Selection and crossover both can't avoid the probability when the search falls into the trap of local maximum value and loses the chance to find the best fitness. Mutation can change one or some genes in a chromosome is another operator in GA. The mutated gene position becomes the opposite value; for example 1 to 0 or 0 to 1 after mutation. Mutation provides a chance for genetic algorithms to escape the local maximum state by creating new gene string.

Figure 2.5 shows the simple and basic working way of mutation. According to the mutation rate the second position of Parent 101111 was picked up to do the mutation. The gene in second position of parent became the opposite value 1. After the mutation, the fitness became 63.

As the crossover, mutation also has a mutation rate M_r to control the amount of mutation in the recombination of each generation. The mutation rate M_r represents how often mutation is performed in a generation. To compare with the crossover rate, any bit in each chromosome has the chance to do mutation. To show the influence of the mutation in each generation, here we use the example from [40] again. Assume there are 30 chromosomes in the mating pool, the mutation rate M_r=0.1 and the chromosome length=30, there are about ninety times (population size* chromosome length *M_r=30*30*0.1= 90) mutation performed in each generation.

In the following, we will investigate the performance of the mutation. To avoid the influence of the crossover, the crossover rate will be set to zero and mutation rate ranges from 0 to 1 in steps of 0.1, 100 different starting populations will be created to do the same evolution. In Figure 2.6 each curve is the average of 100 different starting populations.

From Figure 2.6 it shows that the smaller mutation rate the higher fitness found in this optimisation problem. It means that the small mutation rate is a good choice for SGA such as around 0.1 in this experiment. Too many mutations means the higher possibility of the useful gene becoming lost in the existing chromosomes. In this experiment, SGA to find the more than 80% best fitness in 50 generation in most of time when mutation rate is between 0 and 0.1. In another hand, it shows that the inappropriate mutation rate leads to a very poor performance.

2.1.6 The Evolutionary Mechanism of Genetic Algorithm

In Algorithm 1 the pseudo code of the genetic algorithm describes the basic working way of genetic algorithm. Genetic algorithm repeats the search and operators until the finish result is satisfied. Now we explore how GA realises each step in Algorithm 1.

Algorithm 1 clearly shows that the first step of the GA is a suitable and correct way to encode the optimisation problem to the chromosomes. A population which has a number of chromosomes will be initialised. The fitness of each chromosome in the population will be evaluated by the fitness function. According to the fitness evaluated, some chromosomes

which have the higher fitness will be chosen to the mating pool. The evaluation will be carried on by repeating the main three operators: selection, crossover and mutation until the requirements are met.

Algorithm 1 Standard Genetic Algorithm

1: Choose a encoding way for chromosome
2: Initialize population
3: Evaluate population
4: **repeat**
5: Select the higher fitness chromosome to mating pool according to the fitness of individuals
6: Crossover the genes of selected parents
7: Mutate the mated population stochastically
8: Evaluate the fitness of the new population
9: **until** The finish requirement meet

2.2 The SGA performance

In GAs research, Goldberg spent a considerable effort in analysing the operators of GAs and their effect on performance. To demonstrate the performance of GAs and provide a robust foundation for future research work, this section will reproduce the results of the standard genetic algorithm example [40] with the same operators and parameters. To convince the correctness of the code, the result of running code will compare with this benchmark. In this example the fitness function is the power function same as in last section example.

The following are the parameters used in this experiment.

Population Size =30

Chromosome Length=30

Mutation Rate=0.0333

Crossover Rate=0.6

Figure 2.7 shows the evolution behaviour of the maximum fitness and the average

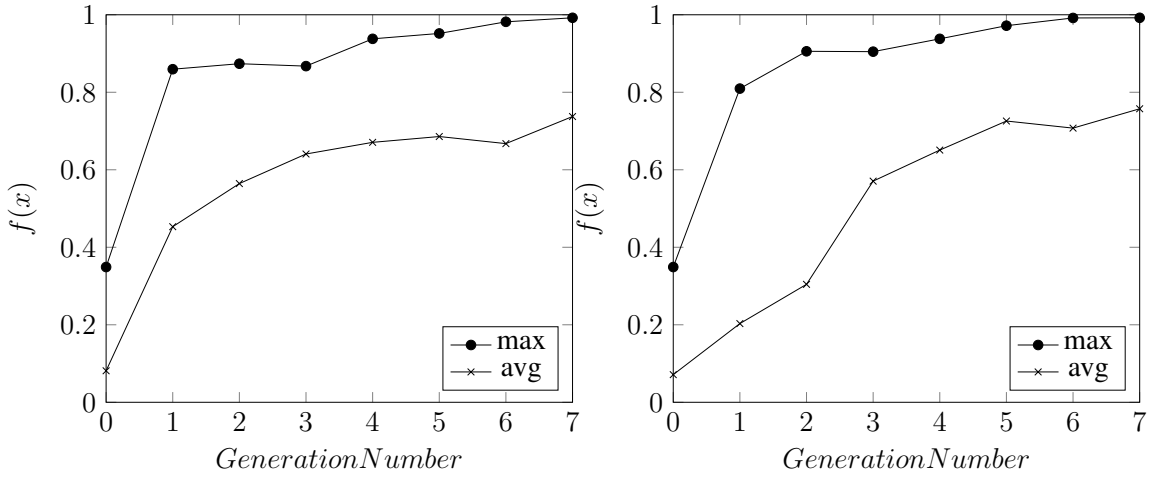


Figure 2.7: standard genetic algorithm fitness evolution

fitness of 30 chromosomes in each generation. The X axis presents the generation number and the y axis presents the fitness f(x). So there are two lines in each graph, one line presents the maximum fitness of 30 chromosomes in each generation and another presents the average one. The left side is the result of this paper and the right side is the Goldbergs result. In comparing these results with Goldbergs, the values and the change trend of fitness matches the Goldbergs. Since the original population is randomly created in this experiment, the fitness value of each plot in this paper is not the exactly same as Goldbergs, but the fitness improve trend is the same. It shows that the genetic algorithm coding in this paper is feasible. Figure 2.7 shows the genetic algorithm can approach the best solution quickly whatever the original population is. The evolution slows down the speed of approaching the best solution, and the evolution approach 90% of the best value in a few steps. It means that the genetic algorithm is suitable for finding an optimal value quickly.

2.3 The Performance of Different Starting Population

In genetic algorithms, the starting population is a considerable factor for the evolutionary speed, as with the crossover rate and the mutation rate. To check the influence of the fitness value in the starting population of genetic algorithms, two different starting populations (set to high fitness and low fitness) were applied to optimise the same function $f(x) = x^{10}$.

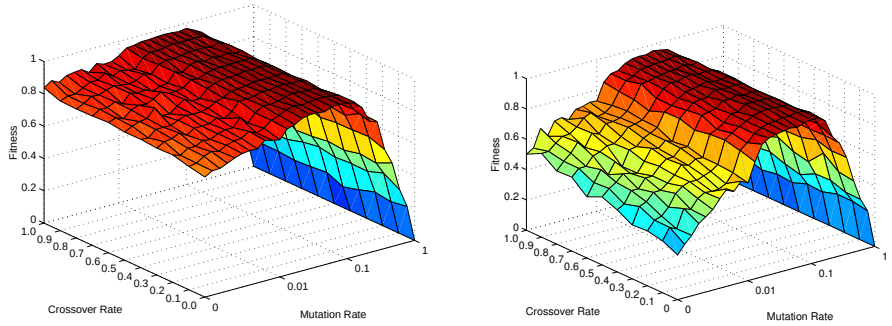


Figure 2.8: The evolutionary speed comparison with different starting populations.

The size of both starting populations were 30 chromosomes. In Figure 2.8, the X axis is the mutation rates and the Y axis the crossover rates. The Z axis is the best fitness after 50 generations with different mutation rate and crossover rate. The fitness of the starting population of the left-hand graph is lower than 0.1. The fitness of the starting population in the right-hand graph is randomly generated between 0 and 1.

Figure 2.8 shows that the genetic algorithm with a starting population with high fitness could approach better fitness than the one with a starting population with poor fitness. It demonstrates that a suitable starting population for a genetic algorithm could lead to a more rapid approach to best fitness.

2.4 Conclusion

This chapter illustrated the feature of standard genetic algorithm by introducing the main operators, the selection policy, and its flow chart step by step. From the investigation and experiments of this chapter the standard genetic algorithm has the following features:

No matter what the complexity or extra information of the problem, genetic algorithm, which does not need other auxiliary information, select the binary encoding string or others encoding strings as an optimisation object and do the evolutionary base on fitness. The genetic algorithms are flexible to optimise various problems with such a mechanism and don't need to worry about the information which is provided by optimisation problem.

Genetic algorithm applies the parallel search strategy rather than step search as other

search algorithms would do. It means that genetic algorithm does not search the optimal result step by step but selects the final target as a research object. The search in genetic algorithm is connotative parallelism, because genetic algorithm explores hundreds or thousands of solution in one generation.

The genetic algorithm has a good search ability. In our experiments, the standard genetic algorithm finds optimal fitness quickly, but approach the best fitness slowly. In section 2.1.5 the genetic algorithm can find more than 80% best fitness in a few generations even if the mutation is turned off. Similarly the genetic algorithm can find more than 70% best fitness in a few generations when the crossover is turned off in the experiment of section 2.1.6. In the experiment in section 2.2 genetic algorithm approaches 95% best fitness in just 7 generations.

According to the definition of the operators of genetic algorithm, it shows that crossover gains the outstanding offspring by exchanging the best or better gene slice in parents. And the mutation obtains a favourable gene slice or individual for the next generation by changing some or one position gene in the parents. An appropriate pair of crossover and mutation rate greatly improves the searching speed for the optimization problem. Finally, it shows that the quality of the starting population also could directly affect the evolutionary speed in genetic algorithm. In another word, a proper starting population leads to a more efficient and high evolutionary speed.

Chapter 3

Constraint Programming

Constraint Satisfaction Problems (CSPs) [65, 99] are mathematical problems which find one state or a solution that is the assignment of values for every variable in such a way as to satisfy all the constraints. It has practical significance because CSPs are general problems that arise in many areas, such as configuration, networking, resource allocation in scheduling and other combinatorial problems [60]. Therefore the constraint satisfaction problems are worth studying because in reality many problems can be modelled as CSPs.

Constraint Programming (CP) [34, 58, 92] is a branch of Artificial Intelligence (AI) used to help us solve CSPs. Many reasoning strategies and search algorithms were proposed and applied in CP for solving the CSPs. The selection of suitable constraint inference strategies and search algorithm directly affects the speed of finding the results of the constraints satisfaction problems.

Therefore there are three main research trends in constraint programming to help it speed the searching. The first one is to develop some efficient inference strategies (sometimes called constraint propagation strategies) which could greatly reduce the searching space and remove those invalid parts such as the invalid value range of variables. The second one is to improve the efficiency of those search algorithms which were applied in constraint programming and could find the solution quicker. The last one is to find a suitable constraint model [35], which could accurately and efficiently describes the CSPs

in some programming languages.

It is obvious that the aim is not only to find the solution(s) of CSPs when the researchers applied constraint programming to solve CSPs, but also to solve it with increasing accuracy and efficiency [35]. To achieve this goal, a series of decisions need to be faced as all those mentioned above. It is time consuming and a burden for the researcher to understand or explore all the strategies and their combinations in constraint programming. A constraints solver [15, 21, 38, 59] is a software system that includes most of popular constraint programming strategies and allows more general and friendly programming language for modelling. The researchers could utilise it to find the results of constraints satisfaction problems by picking up the suitable constraint inference strategies and search algorithm.

The aim of this chapter is to understand the constraint solver and those working principles and strategies applied behind before tuning the constraint solver in this thesis. Therefore this chapter firstly mentions the concept of the constraint satisfaction problems and the constraint solver. In section 3.2 some classic and some state of art constraint propagation strategies in the constraint solver are demonstrated. A classic search algorithm the backtracking algorithm was mentioned in section 3.3. Also a global constraint Alldiff are introduced in detail to enrich the statement on the propagation algorithms. Watched Literals [57] is mentioned as a strategy for implementing global constraints efficiently. The backtracking memory management in Minion is also introduced to illustrate the efficiency of the Minion. Next, four constraint satisfaction problems [30, 101, 47] are introduced before the relevant experiments in this thesis.

3.1 Constraint Satisfaction Problems

To understand the constraint satisfaction problems and constraint solver clearly, the following will introduce the meaning of the constraint satisfaction problems and the value of the constraint solver.

3.1.1 Constraint

Definition 3.1.1 *Constraint*

The constraints decide the possible assignment of values to the variables. A constraint $C(x_1, ..., x_k)$ *represent the logical relation among variables*$x_1, ..., x_k$*where* $k \leq n$*. The variable k which is the amount of variables involved (restricted) is called the constraint arity of* C*. [106]*

According to the amount of the variables involved (or the constraint arity), the common constrains in problem are divided into three types: unary constraints, binary constraints and higher order constraints [88, 94].

Unary constraints: A unary constraint is the domain restriction for single variable such as $x_1 \neq 1$.

Binary constraints: A binary constraint describes the relationship between two variables such as $x_1 \neq x_2$.

Higher order constraints: A higher order constraint introduces the relationship among more than three variables such as $x_1 + x_2 = x_4 - x_3$. High order here means several variables. In another word, a higher order constraints is not a single or straight relationship in/between one or two variables but a more complicated relationship between more than two variables.

In many real problems such as scheduling and resource allocation, it is not always necessary to satisfy all the constraints. Following the restrictions imposed on variables, the constraints can be separated into hard constraints and soft constraints[63].

Hard constraints: A hard constraint gives restricted rules which variables must follow such as $x > 5$.

Soft constraints: A soft constraint is a preference whose satisfaction is not be certain such as Adam prefers more vegetable than meat in his dinner.

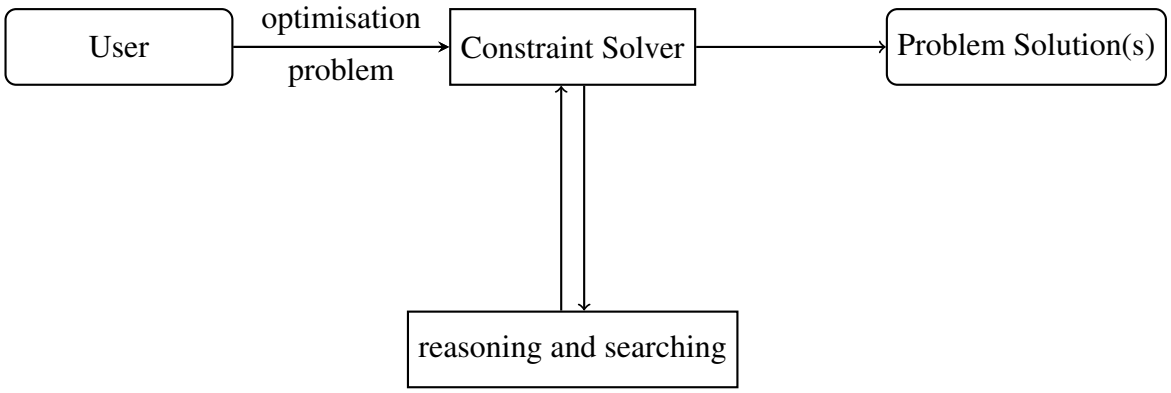


Figure 3.1: The General Framework of A Constraint Solver

3.1.2 The Definition of Constraint Satisfaction Problems

Definition 3.1.2 *Constraint Satisfaction Problems*

Constraint satisfaction problems commonly consist of the following three parts:

1. A finite set of variables $\{x_1, ..., x_n\}$*, where* n *is the number of variables.*

2. Each variable X_i *has a finite domain (value range)* D_i *where* i *is the number of variable and* $i \leq n$*.*

3. A set of constraints (variables relationship) C_m *which restrict the values so that the variables can simultaneously take, where* m *is the amount of the constraints. [92]*

A CSP consists of a set of variables where each variable has their finite set of possible values, and set of constraints so that restrict the values that the variables can simultaneously take. Those possible values of each variable could be consecutive numerical values or individual words such as D_1= {1,2,3.....10} or D_1= {Black, Red.....Blue}. In each finite domain of variables, not all possible assignments of values to variables are permissible.

Although the constraint type could be defined to different variety following the requirements of the optimisation problem, the aim of the most CSPs is [99]:

1. find one solution that is the assignment of values of every variable in such a way that all constraints were satisfied.

2. find all solutions

3 find some optimal solutions

To understand the definition of the CSPs, some examples are given in the following section.

3.1.3 Constraint Solver

As mentioned, a constraint solver is a systematic software which helps a user find one or some of the solution(s) of constraint satisfaction problem. Modern constraint solvers provide a black-box procedure that could solve CSPs with different variables and abundant constraints [42].

In essence, constraint solvers provide a generic combinatorial reasoning and search platform which is automatically generated and hidden from the users as in Figure 3.1.2. The general framework figure has demonstrated that the users only faced the appropriate higher-level representation language to express the problem and pick up the suitable combination of reasoning and searching.

Nowadays the success of the constraint solvers have been proved in many real-world instances. Minion, which is an open source software, is one of the most successful constraint solvers. Because Minion uses a neat problem description way and an expressive input language, it is one of the fastest and most efficient constraints solver [37]. Therefore Minion is selected as one benchmark of the constraints solvers for verifying those tuning algorithms in this thesis.

Minion [36] is one of the general purpose constraint solvers. The pseudocode of Minion demonstrates its working principle for constraints satisfaction problems. Those variables in the CSPs are attached their relative constraints at the beginning of solving CSPs. Then a validation loop will be produced to find the solution(s). In the loop those variables are firstly assigned some values in their domain. Then their constraints will be pushed into a queue. When the constraints are defined multiple variables (more than one), the constraints and the relative variable will be pushed in the queue.

According to some reasoning strategies, all the invalid value of the variables and constraints will removed from the queue. In another word, those variable which have

assigned the values and their constraints will be removed from the queue if their values didn't match their constraints by reasoning. This validation will carried on until empty or solutions are found. Then the validation loop will repeat if the queue is empty but solutions are not found.

Algorithm 2 Minion Pseudo Code

```
Constraints attaches variables;                          ▷ Preprocessing
repeat
    Variables ← Values;
    Pushes constraints or variable into queue            ▷ Preprocessing level
    Propagates(reasoning) until empty                    ▷ Heuristics
until Until solutions found
return Solutions
```

There are four different variable types in Minion: Bool variables, Discrete variables, Bounds variables and Sparse bounds variables.

Bool Variable is a boolean variable which has the domain $[0, 1]$. Bool variables are widely used for logical expressions.

Discrete Variable is one type of integer variable whose domain range from the lower to upper bounds specified. It allows any subset of the domain to be represented.

Bound Variable is also one type of integer variables which have the upper and the lower bounds in their domains. However the domain can only be reduced by changing one of the bounds during the search.

Sparse Bounds Variable is nearly the same as the bound variable, the only difference is that the domain in the sparse bounds variable is composed of discrete values.

The general framework of the constraint solvers and the pseudo code were briefly described and demonstrated in this section. The following will discuss the main solution techniques in modern constraint solvers: reasoning and searching. It will clearly illustrate

how those reasoning strategies and searching algorithms were applied to reduce the domain of the variables and to find the solution(s) in a modern constraint solver like Minion.

3.2 Constraint Propagation Algorithms

Constraint propagation [17], which is also called constraint reasoning or constraint inference, is a form of inferences to reduce the searching space in the CSPs. [29] describes the main idea of constraint propagation is to detect and remove the inconsistent variables assignments with the repeated analysis and evaluation of the variables, domains and constrains.

Local consistency [7] is the most common constraint propagation strategy in constraint solvers. The local consistency includes node consistency, arc consistency, path consistency and so on. Meanwhile, some modern constraint solvers like Minion also generate some global propagation algorithms such as alldiff to reduce the values domain or/and constraints.

3.2.1 Node Consistency

Definition 3.2.1 *Node Consistency*

(x_i, a) *is node consistent if* a *is permitted by* $C_{x_i}(namely, a \in C_{x_i})$ *where* a *is a value of* x_i*. Variable* x_i *is node consistent if all its domain values are node consistent. In another word, node consistency requires that all the values in the variable* $x_i's$ *domain satisfy all its own unary constraint. [74]*

From the definition, it shows that the node consistent could help to move out the unary constraints and reduce the variable's domain.

Example:

Assume the domain of x is $[1, ...10]$ and the unary constraint is $x < 4$.

$x < 4 \& \{x; 1 \leqq x \leqq 10\} \Rightarrow \{x; 1 \leqq x \leqq 3\}$

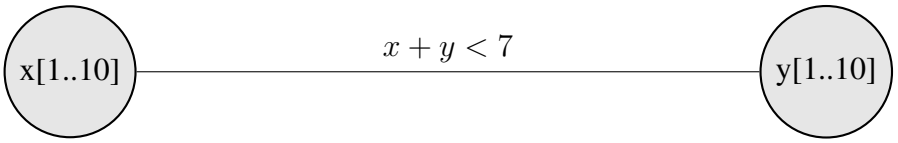


Figure 3.2: The Simple Constraint Satisfaction Problems

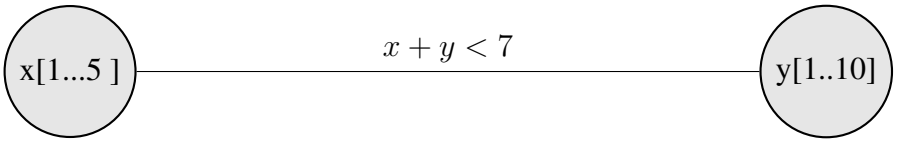


Figure 3.3: The Simple Constraint Satisfaction Problems with $arc(x_i, x_j)$

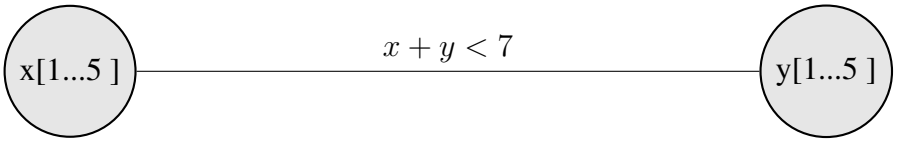


Figure 3.4: The Simple Constraint Satisfaction Problems with $arc(x_j, x_i)$

According the definition of node consistency, the domain of the variable x is restricted to $\{1,2,3\}$ with the constraint $x < 4$ and then the constraints could be discarded.

3.2.2 Arc Consistency

Definition 3.2.2 *Arc Consistency*

If there is a binary constraint C_{ij} between the variable x_i and x_j, then the $arc(x_i, x_j)$ is arc consistent if for every value $a \in D_x$, there is a value $b \in D_j$ such that the assignments $x_i = a$ and $x_j = b$ satisfy the constraint C_{ij}. [84, 99]

Arc consistency (AC) reduces the domain of x_i by removing all the value $a \in D_i$ that couldn't find such a value $b \in D_j$ to satisfy the constraint C_{ij}. The following three figures demonstrate the strategy of the arc consistency with the simple constraint satisfaction problem mentioned before. Figure 3.2 is the constraints network [68, 78] of the simple constraint satisfaction problem. Figure 3.3 clearly shows how arc consistency could be applied to reduce the variables domains in the simple constraint satisfaction problem. Therefore for each value in D_x could find a matched value in D_j which satisfy

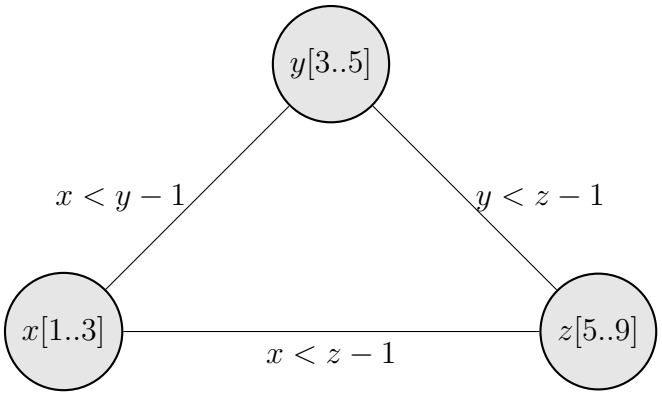


Figure 3.5: Non-path Consistency Example

the constraints. On the other hand, we commonly reduce the D_j with the constraint C_{ji} as in figure 3.4.

Arc consistency is another propagation strategy widely applied in the constraint solver. To efficiently meet the various requests of different problems, some variants of the arc consistency algorithms such as AC-3 and AC-4 [28] have been proposed. In Minion the flag, which provides the option of various AC algorithms, was called preprocessing. It will be mentioned in Chapter 4.

3.2.3 Path Consistency

Path consistency, which is similar to the arc consistency but with more variables involved, can always make further reductions from the constraints for the CSPs.

Definition 3.2.3 *Path Consistency*

The path (X_i, X_j, X_k) is path consistent if and only if for every pair of values $\alpha \in D_i$ and $\gamma \in D_k$ which is satisfied the constraint C_{ik} there is a value $\beta \in D_j$ such that $(\alpha, \beta) \in C_{ij}$ and $(\beta, \gamma) \in C_{jk}$. [68]

From the definition, it shows that the aim of the path consistent is to remove pairs of invalid values $\alpha \in D_i$ and $\gamma \in D_k$ instead of reducing the domains of variables as in node consistency and arc consistency, because there is no such valueβ can be found in D_k which is simultaneously consistent with α and γ.

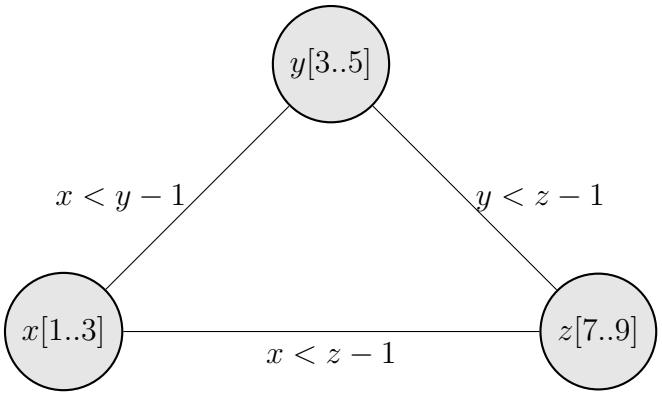


Figure 3.6: Path Consistency Example

In Figure 3.5, the problem mentioned is arc consistent. Each pair of variables meets the requirement of arc consistency. However the problem is not path consistent. When $\alpha = 3 \in D_i$ and $\gamma = 5 \in D_k$ which is satisfied the constraint $C_{ik} = x < z - 1$, there is no value $\beta \in D_k$ such that $(\alpha, \beta) \in C_{ij} = x < y - 1$ and $(\beta, \gamma) \in C_{jk} = y < z - 1$. Therefore the pair of values $\alpha = 3$ and $\gamma = 5$ should be moved out. Figure 3.6 illustrates a proper path consistent problem.

3.2.4 Alldiff

Besides the local consistency, the constraint solves also implement some global constraints such as Alldiff [89] in Minion to reduce the value domain of the variables.

Definition 3.2.4 *Alldiff*

Let $x_1, x_2, ..., x_n$ be variables with respective finite domains $D(x_1), D(x_2), ..., D(x_n)$. Then

Alldiff $(x_1, x_2, ..., x_n) = \{(d_1, d_2, ..., d_n) \mid d_i \in D(x_i),\ d_i \neq d_j$ *for* $i \neq j\}$.

From the name and definition it shows that Alldiff is such a constraint that any two of the variables can't be assigned the equal value.

	0/1 a	0/1 b	0/1 c	0/1 d			0 a	0/1 b	0/1 c	0/1 d
Triggers:	↑	↑			$\Longrightarrow$	a assigned false		↑	↑	

Figure 3.7: Watched Literals Example

3.2.5 Watched Literals

The Watched Literals (WLs) is a state-of-art and efficient mechanism to implement global constraints. In [57] Gent implemented WLs in Minion and showed its importance and efficiency. The WLs was originally applied in the Boolean Satisfiability (SAT) Problems [113] which is a subset of the CSPs.

In traditional WLs two literals called watched literals were assigned in each clause [27]. Those two watched literals could be any two non-false or unassigned literals. Once one of the watched literals was assigned false, the WLs will automatically choose any non-FALSE literal to replace the watch.

Assume there is a clause $a \vee b \vee c \vee d$, all the literals a,b,c and d are unassigned as in Figure 3.7. Therefore a and b could be appointed as the watched literals. If a was assigned false, any non-false or unassigned literals such as c could be the watched literal instead of a. In a word, when a watched literal is assigned FALSE, the WLs attempts to shift the watch to any non-false, unwatched literal in the clause if one exists by searching through all literals in the clause. Meanwhile nothing happens on the watched literal if any other variables were assigned.

One of the advantages of WLs is no cost if a literal is not watched. It means there is no cost when any unwatched literal was assigned. Another is that no cost on backtracking which is a search method and will be introduced later. In another word, there is no change on the watched literals after backtracking.

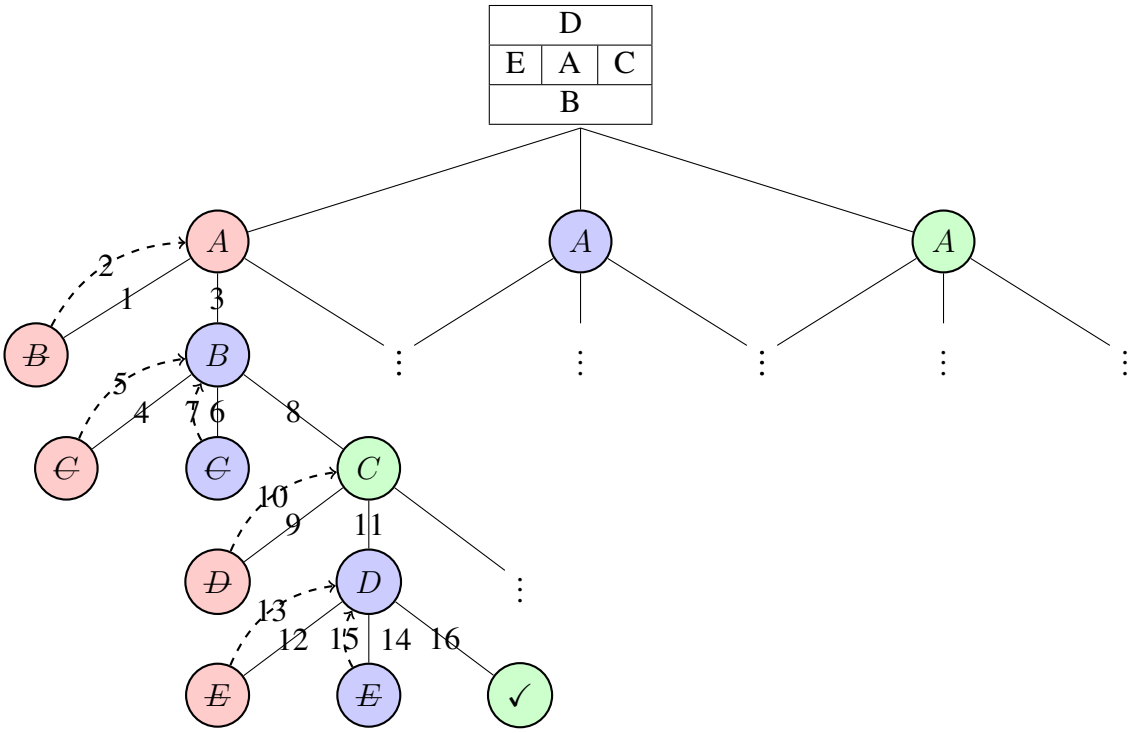


Figure 3.8: Backtracking Algorithm

3.3 Search Strategy

After reducing the search space (domains) and the constraints, a suitable and efficient search strategy is required to find the final solution for constraints satisfaction problem. The general strategy that explores all combination of values to variables is easy to implement but abundant in searching. Therefore some search strategies [46, 83, 97] were proposed to reduce the searching time. In constraints satisfaction problems, a simple backtracking algorithm [92] is the most common and classic systematic search algorithm.

3.3.1 Backtracking Algorithm

Backtracking is a common algorithm for finding the solutions in real world problems. Backtracking Algorithms implement the search strategy. When the current variable is assigned a value from its domain, the backtracking algorithm checks against the constraints between the current variable and the past variables. The current variable will be abandoned if the any of the constraints checked are not satisfied. The search will then go back to

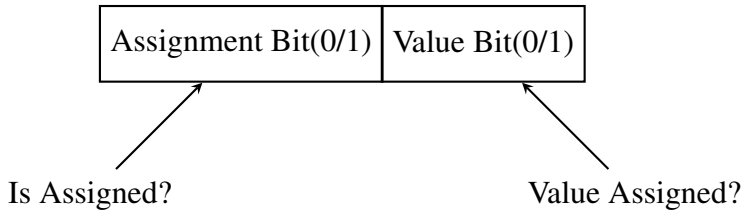


Figure 3.9: Boolean Variables Representation in Minion

the checking between the current variable which is assigned another value and the past variables. If all values in the current variable are tried, the previous variable will be assigned a new value from its domain.

Graph (map) colouring problem [61, 62] is a classical constraint satisfaction problem which concerns on combinatorial optimisation. The graph colouring problem attempts to filling a few of plots in graph with a fixed number colours so that no adjacent plots have the same colour. The Figure 3.8 shows how to fill up with three colours: Red, Blue and Green with the backtracking algorithm.

3.3.2 Backtracking Memory In Minion

As mentioned above, the backtracking is an important part in solving CSPs. It requires enough memory to store the variables. However Minion implements a backtracking memory management approach to solve the memory issue. To occupy less memory and reduce the amount of work during the backtracking, the variables in Minion were divided into two parts: a backtrackable part and a non-backtrackable part.

As in Figure 3.9, two bits were selected to store a boolean variable in Minion. The first bit is an assignment bit which indicates whether the variable is assigned. The second bit is a value bit which indicates which value is assigned. Therefore there are four statues for a boolean variable as in Table 3.1.

If the variable is assigned, the assignment is set to 1. Meanwhile the value bit is set to a value. When the backtracking happened, the assignment bit is reset to 0. The value bit becomes irrelevant until the variable is reassigned. Therefore the value bit does not

Assigned True	Assigned False	Unassigned True	Unassigned False
11	10	01	00

Table 3.1: Boolean Variables Representation in Minion

need to restore. When the variable is unassigned, the value is of course unused. Hence, the assignment bit requires a backtrackable block to store. The value bit is stored in a non-backtrackable block. Such a memory assignment approach greatly improves the efficiency of the memory usage.

3.3.3 Variable Ordering

"To succeed, try first where you are most likely to fail." [50]

The above idea is one of the variable heuristic ways in CSPs. A search method in CPS requires the order in which variables are to be assigned to be specified. The correct order of variables can noticeably improve the searching efficiency of CP. In Minion there are eight variable orderings called varoder for the search process: sdf [13, 14, 100], sdf-random, srf, srf-random, ldf, ldf-random, random and static [32].

sdf - sdf is the aberration of the smallest domain first. It attempts to select the variable which has the smallest domain to assign the value firstly. The sdf break ties lexicographically when the domain size of a few variables is the same as the smallest.

sdf-random - sdf also attempts to select the variable which has the smallest domain to assigned the value firstly, but breaks ties randomly.

srf - Smallest ratio first (srf) chooses the unassigned variable with smallest percentage of its initial values remaining. When more than one variable has the ratio, the srf breaks ties lexicographically.

srf-random - srf-random is almost the same as the srf. The srf-random chooses the unassigned variable with smallest percentage of its initial values remaining firstly. However the srf-random breaks ties randomly.

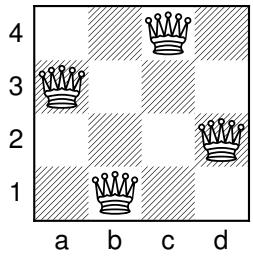


Figure 3.10: Four Queens Problem

ldf - To compare with sdf, largest domain first(ldf) picks up the variable which has the largest domain and breaks ties lexicographically.

ldf-random - ldf-random selects the variable which has the largest domain, but break ties randomly.

random -random means that a random variable ordering is implemented.

static - static means that the variable ordering to be assigned is lexicographical.

3.4 Four Constraint Satisfaction Problems

In this thesis, four classical constraint problems were chosen from the CSPlib [1] for Minion tuning: the N-queen problem, the langford's Number problem, Balanced Incomplete Block Design and Golomb Rulers. They will be implemented in Minion to verify the efficiency of our tuning algorithms in the later chapters. The following introduces their basic descriptions before those experiments that they are implemented.

3.4.1 N-Queen Problem

In chess, a queen can move as far as she places, horizontally, vertically, or diagonally. A chess board has n rows and n columns. The standard n by n Queen's problem asks how to place n queens on an ordinary chess board so that none of them can hit any other in one

[1] All from www.csplib.org

move [109]. The search space and running cost dramatically increases with the number of queens in the problem. Figure 3.10 shows a solution of the Four Queens Problem. N-Queens is a classic and tractable problem which has existing constructions.

3.4.2 Langford's Number Problem

The problem generalises to the $L(k, n)$ problem, which is to arrange k sets of numbers 1 to n, so that each appearance of the number m is m numbers on from the last [47]. The computing complexity of finding solution in Langford's number problem depend on the two variables which are different from one variable in the N-Queen problem. Table 3.2 is a solution for L(2,4)

3.4.3 Balanced Incomplete Block Design

A Balanced Incomplete Block Design (BIBD) [85] problem is defined as an arrangement of ν distinct objects into b blocks such that each block contains exactly κ distinct objects, each object occurs in exactly γ different blocks, and every two distinct objects occur together in exact λ blocks. Another way of defining a BIBD is in terms of its incidence matrix, which is a ν by b binary matrix with exactly γ ones per row, κ ones per column, and with a scalar product of λ between any pair of distinct rows. A BIBD is therefore specified by its parameters $(\nu, b, \gamma, \kappa, \lambda)$. An example of a solution for (7,7,3,3,1) is:

0 1 1 0 0 1 0

1 0 1 0 1 0 0

0 0 1 1 0 0 1

1 1 0 0 0 0 1

0 0 0 0 1 1 1

1 0 0 1 0 1 0

0 1 0 1 1 0 0

4	1	3	1	2	4	3	2

Table 3.2: Langford's problem instance L(2,4)

3.4.4 Golomb Rulers

A Golomb ruler [72] may be defined as a set of m integers $0 = a_1 < a_2 < \ < a_m$ such that the $m(m-1)/2$ differences $a_j - a_i, 1 <= i < j <= m$ are distinct. Such a ruler is said to contain m marks and is of length am. The objective is to find optimal (minimum length) or near optimal rulers. Note that a symmetry can be removed by adding the constraint that $a_2 - a_1 < a_m - a_{m-1}$, the first difference is less than the last.

3.5 Conclusion

In this chapter, it describes the state-of-art and important concept and strategy of constraints programming. After the fundamental definition of CSPs, the local consistency, a global constraint Alldiff and a global propagation WLs were discussed. It demonstrates how those different strategies and methods in a modern constraint solver Minion were implemented to solve the CSPs. It also illustrates how Minion implements memory management in backtracking. At the end of the chapter, four different CSPs, which will be implemented in the rest of the thesis, were mentioned in advance.

Whatever the constraint programming technologies chosen, the constraint programming firstly defines the variables and their domains in the problem. Next, it applies the constraint propagation to reduce the domains of the variables or the search complexity and to remove or reform the constraints. Finally it applies an efficient search mechanism to find one or all solutions. Therefore the combination of the proper reasoning approach, the suitable modelling and the appropriate search mechanism becomes to the key of improving the efficiency of the constraint solver.

According to understanding the feature and skill of constraint programming, it provides a knowledge and inspiration to help improving the tuning for the constraint solvers in the

later chapter.

Chapter 4

Standard Genetic Algorithm for Tuning

In the last chapter, some classic definitions, searching methods and reasoning strategies of the constraint programming were introduced to show how a modern constraint solver is constructed. Before finding the solution(s), the parameter tuning for a constraint solver is a complicated job for beginners in the constraint programming area and a time-consuming workload even for the expert in constraint programming [66]. There are two tuning approaches mentioned in the thesis. One is to focus the tuning of one instance. It means there is no other auxiliary information and the aim is to find the optimal or best parameter set(s) for one instance in a specific time limit. Another is to seek optimal parameter set(s) by tuning training instances for unknown testing instances.

In this chapter, we only consider the first kind of the tunings *the single instance tuning*, and a genetic based configuration mechanism for tuning the single instance called GACM is discussed. In section 4.1, it explains the significance of parameter tuning and the reason that chooses the genetic algorithms to help tuning. The genetic based configuration mechanism for Minion is proposed and verified in section 4.2. Section 4.3 demonstrates how to realise the GACM with the standard genetic algorithm. In section 4.4, the efficiency of GACM is verified by tuning various optimisation problems and comparing the tuning performance with the Random selection. The last section is the evaluation and the conclusions for the genetic based configuration mechanism.

4.1 The Significance of the Tuning with Genetic Algorithms

In a constraint solver, various reasoning strategies and search methods could be chosen and implemented for solving CSPs as the parameters. The efficient parameter selection could save the running cost and reduce the search complexity. On other hand, an inappropriate parameter set leads to a redundancy and overlapping searching. The parameter selection problem also happens in comparing two algorithms or modellings. It is obviously unfair to compare one algorithm with a correct parameter setting with another with an improper setting. Therefore the tuning is a vital factor in problem optimising or new algorithm exploring. In constraint programming, the job of tuning the constraints solver is currently done manually. It becomes a barrier which slows down the time of exploring new algorithms or finding the solutions of the constraint satisfaction problems.

It shows that many automatic tuning (configuration) approaches have been proposed and widely discussed in the survey part. In chapter 2, the standard genetic algorithm was introduced and explored to show its robustness and efficiency as an optimisation technology. Therefore it is considered and implemented to cope with the tuning problem in this thesis. [5] concluded two advantages of applying genetic algorithms for automatic tuning: One is that genetic algorithms are known to be very robust with respect to optimisation problems that have undesirable objective landscapes [40]. This is due to the influence of tuning the parameters in a constraint solver or for an algorithm is unknown. A robust algorithm is expected to deal with whatever objective landscape we encounter. Another is that genetic algorithms are inherently parallel [20]. In the solver configuration the most time consuming step is to evaluate the running time of any parameters sets for the solver. Genetic algorithms allow parameter sets to compare and race against each other in each generation. Meanwhile the framework of genetic algorithms allows such evaluations to happen in the same time that will great save time in practice. Those features of genetic algorithms will be illustrated and improved step by step in the following chapters.

In this thesis, a few genetic algorithms about the single instance tuning are firstly

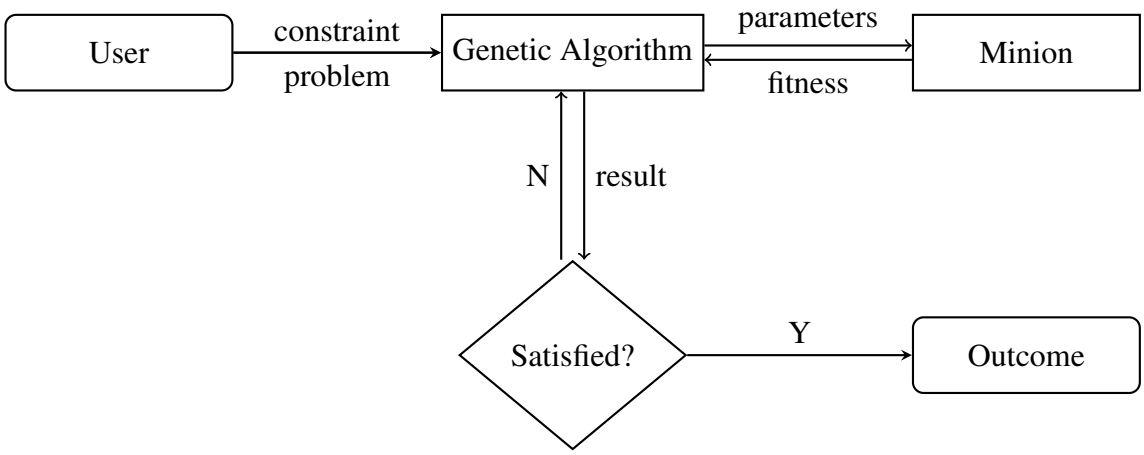


Figure 4.1: The Framework of Genetic Algorithms Configurator for Minion

discussed. There are two motivation on the single instance tuning. Firstly, the single instance tuning verifies that genetic algorithms are feasible to implement in the constraint solvers tuning. It also demonstrated how genetic algorithms successfully obtain an optimal or best parameter set for tuning a specific problem, when the aim of tuning the specific instance is to find not the result of the instance but the optimal or best parameter. Secondly, the idea in the single instance tuning provides a reasonable basis for the instance based tuning in the later chapters.

In this chapter, it will illustrate how the standard genetic algorithm is implemented for tuning a constraint solver. A genetic based automatic configurator for Minion named $(GACM)$, which could suggest an optimal parameter set for a given constraint satisfaction problem, was proposed and verified. The following will firstly present the working mechanism of the genetic algorithms in $GACM$.

4.2 The Framework of the GACM

Genetic based automatic configurator for Minion is a configurator which implements the standard genetic algorithm to help Minion find an optimal parameter set for a given instance in a specific time limit. Figure 4.1 indicates the framework of genetic based automatic configurator for Minion.

According to the optimisation problem, some parameter (switch) sets, which present the searching methods, reasoning strategies and/or modelling approaches to implement, were randomly initialised as the chromosomes. The standard genetic algorithm sends those suggested parameter sets (chromosomes) to Minion, and get the running cost of each parameter sets back to the genetic algorithm. Then the genetic algorithm pushes those candidates (each set of switches) which have perfect fitness (less running cost) into mating pool. After crossover and mutation, the new suggested switches will send to Minion to evaluate the running cost. Meanwhile the best switch setting for Minion in each generation will be recorded. The GACM will repeat the evolution until the best setting was found or the requirements satisfied.

4.3 The GA Design in Automatic Configurator

After understanding the framework of the GACM, the following explains how the standard genetic algorithm is implemented and adapted to help tuning in the genetic based automatic configurator.

4.3.1 Encoding

Encoding in genetic algorithms is to transfer solutions of optimisation problem to the chromosomes that each chromosome presents one possible solution. The aim in GACM attempts to automatically hunt the best or some optimal parameter sets for Minion in a specific time limit. Each chromosome in GACM is a parameter set which indicates the prepossessing level, search strategy, modelling method and so on.

In this chapter our automatic genetic configurator attempts to tune three classic flags (parameters). To justify the performance, GACM is also implemented to help tune the modelling. In the flag tuning, there are three switches considered to tune: preprocess(5 values), prop-node (5 values), varorder (8 values) [36].

Flags	x_1			x_2			x_3			
Position	1	2	3	4	5	6	7	8	9	10
Encoding	1	0	0	0	1	0	1	0	0	0

Table 4.1: Encoding format in genetic configurator

switches preprocess The preprocess switch allows the user to choose what level of preprocess is applied to their model before search commences.

switches prop-node The prop-node switch allows the user to choose the level of consistency to be enforced during search.

switches varorder The varorder switch enables a particular variable ordering for the search process.

Each chromosome presents the statues of those three flags in Minion. The following explains how the switches are encoded to the chromosomes in the automatic genetic configurator:

$x_1 = \{0, 1, 2, 3, 4, 5\}$ indicates the five different flags in the preprocess switches and one situation which turns the preprocess switches off. In binary, three bits are required to present the six flags.

$x_2 = \{0, 1, 2, 3, 4, 5\}$ indicates the six different flags in the prop-node switches. It is also required three bits.

$x_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ indicates the nine different flags in the varorder switches. In binary, four bits are required to present those nine flags.

Therefore each chromosome can be constructed as in table 4.1. It means that the chromosome length in the genetic based automatic configurator depends on the amount of the parameters need to tune. Therefore the chromosome length is assigned to ten to present those three switches in GACM.

However, in such a encoding way, some invalid parameter binary string, which presents each switch, may appear when the genetic algorithm initials the starting population or

generates the next generation. For example 111 is an invalid binary string to present x_1 in GACM. When it happens, the binary string will be replaced by randomly choosing a valid value from that parameter.

4.3.2 Fitness in GACM

The fitness function gives the information to the selection process to pick up the parents in a mating pool. In this thesis, we attempt to find an optimal or best parameter set which could find the solution(s) with less running cost for the optimisation problem. The running cost is an ideal scale to judge the quality of the parameter found. However genetic algorithms are used to find the maximum value. Therefore the fitness function in here is

$$F(x) = 1/x$$

where x is the running time of finding the solution with relative parameter sets.

4.3.3 Reproduction in GACM

In GACM, the reproduction includes selection, crossover and mutation which are the same as in the standard genetic algorithm. In our genetic configurator, the selection is the roulette wheel selection [41].

Single point crossover is the classic and most common crossover in genetic algorithms because it can be easily understood and realised. In our thesis the single point crossover will be selected firstly. The two point crossover [103], which swipes the parts between two crossover points, is also explored and compared with the single point crossover in our experiment. In our experiment the mutation rate is the probability that any bit in each chromosome does a mutation. The mutated gene position becomes the opposite value Etc. 1 to 0 or 0 to 1 after mutation.

To avoid the time wasting on evaluating inappropriate parameter sets, we set a cut-off time for each fitness evaluation in each generation. The cut-off time depends on the tuning

instance of each optimisation problem. More details about the cut-off time is discussed in the later section.

4.4 Experiments Design

In this section, some experiments are designed and implemented to show the efficiency of the standard genetic algorithm on tuning the Minion. Those experiments could be divided to three parts. The performance of GACM is firstly checked by applying GACM to deal with some classic CSPs. Then we attempt to check the parameter sensitivity of the genetic algorithms in GACM. Finally GACM compares with the random selection on tuning not only those inherent parameters in Minion but also those models generated by other modelling assistant such as Savile Row [81].

The following experiments were run on a 64 bit Linux Intel Core i7-4790 Haswell Quad-Core, 8 GB RAM, 256 GB solid state drive and 2 GB GTX 960 graphic card. Four classic optimisation problems, N-queens problem, Langford's Number Problem, Balanced Incomplete Block Design(BIBD) and Golomb Rulers, which are mentioned in last chapter, were chosen as the optimised problems.

From the definition description of problems, it shows that those four constraint problems are very different to each other. The computational complexity of N-Queen problem and Golomb Rulers depend on one variable. The complexity of Langford's Number Problem is up to two variables. We hope a genetic based automatic configurator could be feasible to different constraint satisfaction problems.

4.4.1 The Performance of GACM

The aim of the performance testing is to demonstrate how the optimal parameter sets were explored under the evolutionary strategy in genetic algorithms. In the performance testing of GACM, the crossover rate is firstly set to 0.9 and the mutation rate is set to 0.1. In the first performance testing of GACM, we attempt to tune three switches in

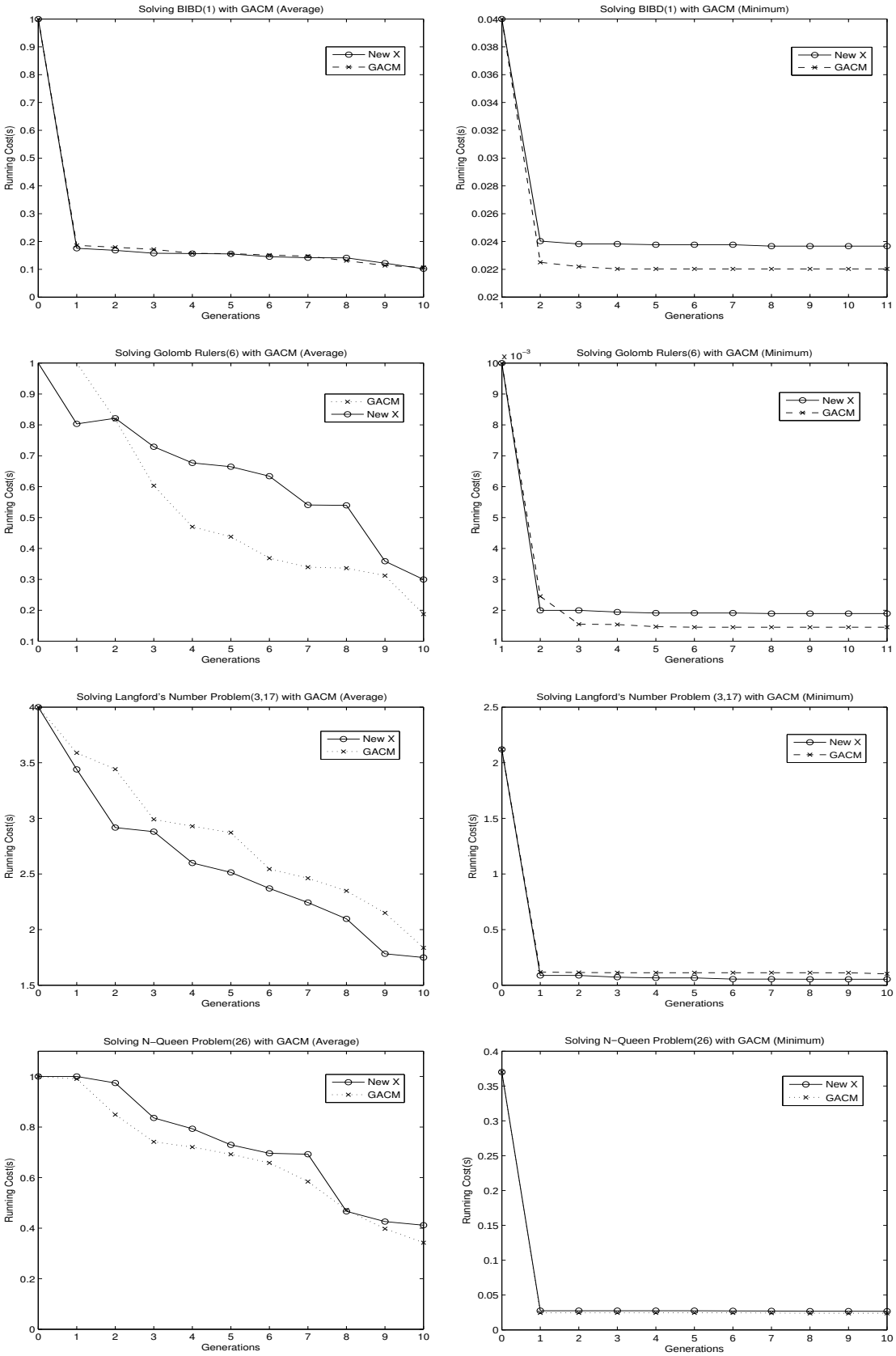


Figure 4.2: The Efficiency of Solving Optimisation Problems by GACM with Standard Crossover and New Crossover

Minion. According to the searching complexity, we only consider ten generations of the GA and the population size of the GA is set to 10. Each trial runs ten times and we observe the average of the minimum and the average. As mentioned, the evaluation on each parameter set is the most time consuming part in the automatic tuning. To avoid the time wasting on evaluating the invalid or bad parameter sets, a specific cut off time is set for each instance. The cut off time for each instance in this chapter is commonly set by the running time, which implemented the default parameter setting, if there is no specific mention. Meanwhile an attempt and evaluation about new crossover (two points crossover) in GACM are implemented to tune the minion with the same instances. Two point crossover applied the same strategy as the standard single point crossover, but it randomly chooses two crossover points in the same time instead of one point.

Figure 4.2 illustrates the performance of GACM on solving fours constraint problems: BIBD(1), Golomb Rulers(6), Langford' Number problem (3,17) and N-queen problem (26). The numbers, which are followed by the problem name and in brackets, indicates which instance is chosen from the problems to verify the performance of GACM. Figure 4.2 demonstrates the average of the minimums and the average of ten times tuning on each instance. In Figure 4.2, the left column illustrates the average of the minimum and the right column is the average of the average. The two curves in the figures demonstrate the performance of GACM with standard crossover and two point crossover. The X axis is the number of generation of genetic algorithm in tuning fours optimised problem with GACM, and the Y axis in the left column is the average of the running cost of finding the solution with all parameter sets in each generation. The Y axis in right hand side figures is the average of the running cost of finding the solution with the best parameter sets of each generation.

The curves in Figure 4.2 indicate that GACM gained a satisfied parameter setting for Minion in solving four optimisation problems after just few generations. In the figures of BIBD(1) the standard crossover could gain a better parameter set, which could find the solution(s) faster, than the two points crossover. However the average figure can't conclude

which crossover is better. In Golomb Ruler(6) the standard crossover outperforms the performance of two point on both the average and the minimum. But in Langford's Number Problem (3,17), the experiment results are exactly reversed. The two point crossover shows its efficiency over the standard crossover in GACM. The standard crossover demonstrates its performance over the two point crossover again in the N-queen problem (n=26). From the Figure 4.2, it can't conclude that the standard crossover is a better choice than the two points crossover in GACM. To make a feasible conclusion, their running time on tuning are recorded to compare as well.

	Standard Crossover	Two-points X
N-Queen (26)	409275ms	427870ms
BIBD (1)	35577ms	35848ms
Langford's Number Problem (3,17)	1379334ms	1442879ms
Golomb Rulers (6)	183086ms	240821ms

Table 4.2: Running cost with different crossover operator

Table 4.2 shows the running time difference in tuning the above four instances with the standard crossover and the two points crossover in GACM. The comparison shows that the two-points crossover always spends more time than the standard crossover, because it costs more CPU time on crossover. Since Figure 4.2 showed that the standard crossover outperforms the two points crossover in most time, it means that the standard crossover is a better choice in GACM.

4.4.2 The Parameter Sensitivity of GACM

In [40] it said that the parameter setting of genetic algorithms itself is very hard to control. A proper genetic algorithm parameter setting will lead to a greater searching speed and vice versa.

In chapter 2 we explored the parameter sensitivity of genetic algorithms in solving Dejong's function. In this section we will explore the parameter sensitivity of GACM for tuning Minion in solving N-queen and Langford's Number problem. The aim of

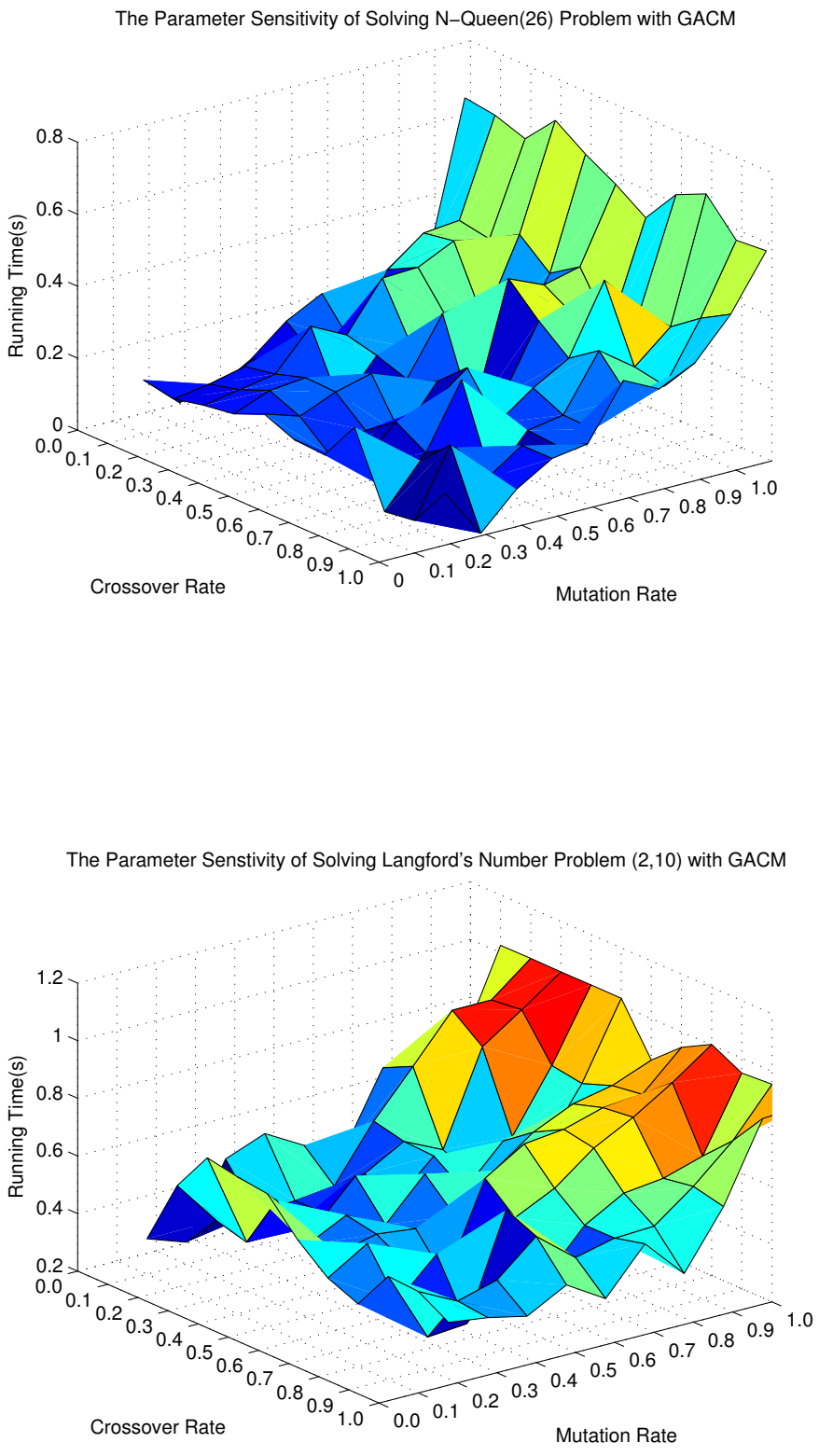


Figure 4.3: The Parameter Sensitivity of Genetic Algorithm in GACM

the parameter sensitivity testing here is to explore the influence of the parameter itself in GACM, and give a cue for initialising those parameters in GACM when GACM is implemented for tuning in the rest chapters.

In the parameter sensitivity testing, N-queen problem (n=26) and Langford's Number problem (2,10) were selected as the optimisation problems since they are two different types of problems. Figure 4.3 shows the exploration on the parameter sensitivity of genetic algorithms in solving those two problems by GACM. In Figure 4.3 the X axis and Y axis are mutation rate and crossover rate of genetic algorithm in solving the Landford's Number Problem and N-Queen problem with GACM. The Z axis is the best running cost of finding the solution with GACM which applies the relative crossover rate and mutation rate in the genetic algorithm in 10 generations.

It is obvious that there is a lot of noise in our experiment in Figure 4.3. But the 3D graphs illustrate that the running cost of solving the optimisation problems become less with the decrease of mutation rate when the mutation rate is between 0.1 and 1. The performance of the crossover is not as obvious as the mutation is. The running cost of solving the optimisation problems reduce with the increase of crossover rate when the crossover rate is between 0 and 0.9. From Figure 4.3, it indicates that the mutation plays a key rule rather than crossover. Figure 4.3 also shows the best mutation rate is around 0.2 and the best mutation rate is around 0.9 in those two experiments.

4.4.3 The Comparison with Random Selection

In order to give further study on the performance of the GACM, the random selection strategy is selected to compare with the GACM in tuning Minion. To compare with the instances chosen in section 4.4.1, each optimisation problem chooses five different instances for tuning.

The population size, mutation rate and crossover rate in GACM are the same as in previous subsection. From the result in the performance of GACM, it shows that the optimisation performance of GACM is obvious and effective in the first few generations.

The best parameter sets for tuning could be more easily found when more parameter sets were evaluated. However it means the tuning has to occupy more CPU computing time. Since we just consider three switches tuning in this paper, it is not necessary to observer the performance of GACM in a lager generations. Therefore the generation size is assigned to three. Each tuning trail for each instance reproduces ten times as in section 4.4.1 for comparing later.

Since the population size is ten and the generation is three in GACM, it means the GACM finds the optimal parameter set by evaluating thirty parameter sets for each instance in each trial. To gain the same amount attempts as in GACM, the random selection randomly selects thirty parameter sets to evaluate each time. The random selection implements such a trial ten times to observe the average performance. Next, GACM compares the average of the minimum and the average running cost with the random selection.

Table 4.3 is the comparison between GACM and random selection over twenty instances. In table 4.3 there are four main columns: instance name, cut off time for each instance, GACM and random selection. Meanwhile there are four sub-columns under GACM and random selection to compare their efficiency: Running cost, Minimum, average and invalid. The running cost is the average running cost for each tuning over ten times trials. Minimum and average means the average of the minimum and average in each tuning trial. The ”invalid” column represents the amount of the parameter sets, which can’t find the solution in the cut off time in each tuning trial.

The result in table 4.3 clearly indicates that the GACM outperforms than the random selection on either the minimum or the average. Meanwhile GACM could spend less running costs than the random selection does because GACM could successfully avoid evaluating more invalid parameter sets.

To verify the efficiency of GACM in further, it compares with the random selection on tuning Minion and the modelling selection in the same time. To compare GACM with random selection in new instances which includes the modelling selection, we could

Instance	Cutoff (s)	GACM				Random Selection			
		Running Cost (ms)	Minimum (ms)	Average (ms)	Invalid	Running time Cost (ms)	Minimum (ms)	Average (ms)	Invalid
N-Queen(26)	1	15858	25	222	10	22358	39	273	20
N-Queen(27)	1	18008	28	287	11	22153	32	269	20
N-Queen(28)	4	52831	26	647	9	85064	48	876	21
N-Queen(29)	2	31051	28	472	10	46315	43	630	21
N-Queen(30)	1	895120	28	15281	9	1042615	36	11770	19
BIBD1	1	1069	24	29	0	1073	27	34	0
BIBD2	1	1467	24	42	0	1326	26	42	0
BIBD3	1	11649	28	59	10	17101	32	246	13
BIBD4	1	10686	27	134	7	16875	31	317	12
BIBD5	1	13098	28	37	12	19300	33	47	19
Golomb(6)	1	5857	1	94	3	14780	2	179	12
Golomb(7)	1	11640	13	85	9	20481	23	203	18
Golomb(8)	1	20372	16	449	9	27804	253	409	27
Golomb(9)	5	135604	1817	3760	9	144145	2404	3405	27
Golomb(10)	35	1017300	1674	28986	10	1034502	1798	27252	28
Langford(2,10)	1	22634	179	487	10	24288	256	477	19
Langford(2,19)	6	78126	27	972	9	120447	33	1232	18
Langford(2,20)	5	63822	28	662	10	103695	30	924	20
Langford(3,17)	4	71353	102	1001	12	100186	448	1421	23
Langford(3,19)	4	90467	95	1419	17	111830	854	1557	27

Table 4.3: The Efficiency of GACM in Solving Different Problems by Comparing the Random Selection

		GACM				Random Selection			
Instance	Cutoff (s)	Running Cost (ms)	Minimum (ms)	Average (ms)	Invalid	Running time Cost (ms)	Minimum (ms)	Average (ms)	Invalid
3-7-7-6	1	28845	59	127	272	37044	114	199	361
3-8-8-7	1	30636	112	248	262	39482	360	419	382
4-3-4-6	1	25710	35	93	243	36454	52	290	346
4-4-3-7	1	23995	32	71	235	36175	38	198	346
4-4-4-8	1	39358	227	492	299	41648	473	589	407
4-4-5-10	1	36594	n/a	n/a	500	43066	n/a	n/a	500
4-5-4-10	1	37334	364	596	317	41081	517	631	394
5-4-3-8	6	239259	1605	2385	400	246795	1967	2517	404

Table 4.4: The Efficiency of GACM in Modelling Selection by Comparing the Random Selection

reproduce the last experiments by only changing the encoding part. GACM only needs to enlarge the length of the chromosomes that the parameter sets have more space to present the modelling options. Equidistant Frequency Permutation Arrays, one of the benchmark in Minion [52], will be selected as the tuning problem. In this benchmark, there are six modelling options. So the length of the chromosomes extends 3 bits to present those six modellings. However other configurations in the GACM itself such as the population size and crossover rate are assigned the same as in the previous experiment.

Table 4.4 is the result of tuning the new instance with modelling selection by GACM and random selection. The columns in table 4.4 are similar to the table 4.3 because the comparison items are both the same. The result in the table 4.4 shows that GACM could achieve a better tuning result no matter on the minimum or the average over the random selection. It also matches the conclusion in last experiment.

4.5 Conclusion

In this chapter we proposed a genetic based automatic mechanism for Minion. The GACM was applied to tuning different problems to check the efficiency. Four problems were selected to test the efficiency of the GACM. Although the tuning ability of the GACM is not as extraordinary as we expect to find the best tuning, it is feasible that the GACM could always achieve much better parameters' tuning. It means the idea and the mechanism of GACM are acceptable in the experiments.

To improve the efficiency of the GACM, a new crossover strategy, which selects two points to do the crossover instead of the traditional one point crossover, is attempted. It is obvious from the experiment result that the two point crossover causes more time than the single point crossover. We expect the more chances of exchanging genes between each pair of parents could lead to a more rapid evolutionary. However the result shows the two point crossover can't do better than the single point crossover in most times. It means that an excessive crossover is unlikely to help in stimulating the evolutionary.

From the results of last chapter, the parameters setting in genetic algorithms itself is a vital factor for the tuning speed. Therefore the influence of the crossover rate and the mutation rate in genetic algorithms was explored in this chapter. In the sensitivity testing of the mutation and the crossover, it shows that mutation has a more important role than the crossover. It matches the feature of tuning that a few flags tuning of better setting would lead to the best flag setting. It means that the flags difference between the best prepossessing level and the better prepossessing level are few. The exchanging parent's part gene information can't lead to a significant influence for the tuning evolutionary as the mutation did. The result suggests that the ideal mutation rate is around 0.1 and crossover rate is around 0.9 in GACM.

Finally, the performance of GACM compares with the random selection in tuning on the processing level and modelling. Since each chromosome (parameter sets) races against each other, good parameter sets have more chance to pass to next generation. Meanwhile the mechanism, to a great extent, has avoided the invalid parameters happening. GACM showed its superiority on time consumption and search result over the random selection.

Chapter 5

Sexual Genetic Algorithm

It has been mentioned that the main idea behind the genetic algorithms is derived from the evolutionary theory of natural selection [23, 39]. The genetic algorithms improve the fitness of each chromosomes with crossover (recombination) and mutation [45]. The selection is to choose more fitting individuals (chromosomes) for replacement or mating [98]. It is also one of the most important parts in genetic algorithms. In chapter 3 two classic selection strategies, the roulette wheel selection and the tournament selection, were introduced and investigated. This chapter discussed two more selection strategies: the elitism replacement policy [26] and the sexual selection [86, 96].

Thus section 5.1 considers the efficiency of the elitism replacement policy by testing two different functions. Next, a sexual selection is introduced in section 5.2. To improve the efficiency, a parallel mechanism in the genetic algorithms is discussed in section 5.3. In section 5.4 a sexual genetic algorithm is proposed and verified by comparing with the gender genetic algorithm in tuning Minion and SAPS. Finally, the last section concludes the performance of the sexual selection strategy in further.

5.1 Elitism Replacement Policy in Genetic Algorithm

In [11], it is shown that the selection could control the level of exploration or exploitation by balancing the genetic diversity [105] and selective pressure [112]. The selective pressure

in each generation of the genetic algorithms is a tendency to select the best chromosomes (individuals) of the current generation to propagate to the next generation. This kind of selective pressure is caused by the evaluation and the selection of the fitness function. In biology the genetic diversity commonly refers to the vast number of genetic characteristics or the huge diversity in a species [104]. In the genetic algorithms the genetic diversity means a diverse solution (individual) population.

The selective pressure and genetic diversity interacts in the evolutionary. The high selective pressure means the little genetic diversity which easily leads to a premature convergence [111]. In other words, the high pressure may fall to the local optimal trap. The low selective pressure means that the population could have a rich genetic diversity which keeps vast different genetic characteristics [33]. It means that the low selective pressure consumes more time to converge into a global optimum.

To balance the selective pressure and genetic diversity, many selection strategies were posed and applied. The following section discusses two new selection mechanisms: the Elitism Replacement Policy and the Sexual Selection.

5.1.1 Elitism percentage testing with easy function

The Elitism Replacement Policy is the strategy to select the percentage selection of the best or most fit chromosomes which directly imitate from parents to the offspring. The elitism percentage means that many good fitting chromosomes are directly kept in the next generation. In other words, the elitism percentage prevents the loss of the best or most fitting chromosomes in the old population, and helps them be moved from the old population into the new population. It means that the Elitism Replacement Policy could help improve the selective pressure towards to a greater fitness. However it was mentioned that the genetic algorithms should avoid the premature convergence which was caused by high selective pressure.

Section 2.1 introduced the point that there are three main operators in the genetic algorithms: crossover, mutation and selection strategy. The efficiency of the crossover

and mutation was discussed in Chapter 2. It shows that the offspring were generated by crossover and mutation after the mating parents' were chosen. But the elitism mechanism, which could protect the elitism in the parents generation completely pass to the new generation as the offspring, wasn't discussed in the standard genetic algorithm.

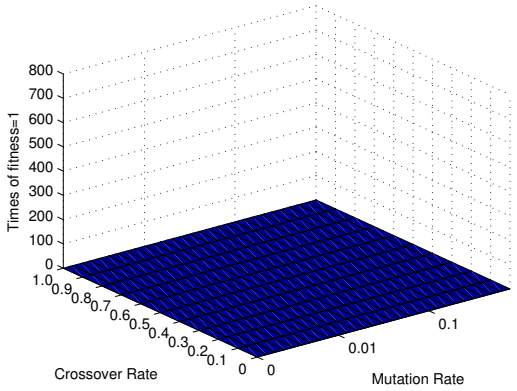
The following research focuses on exploring the suitable elitism percentage for genetic algorithm. The elitism percentage will be tested from 0% to 100%. The experiment implements the same fitness function as in [40].

$$f(x) = x^{10}$$

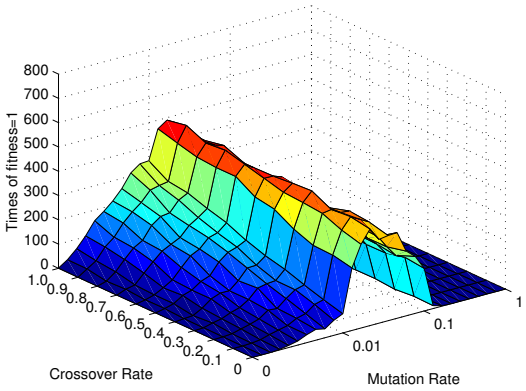
Therefore the parameter settings in the experiment are also kept the same as in [40]: chromosome length is 30, population size is 30, mutation rate is ranged from 0.0 to 1.0, crossover rate ranges from 0 to 1, and generation is 50. Plot average fitness values use 1000 of the same starting populations. Each plot means the average of 1000 starting populations of each population's average fitness at each generation. Again one is the target fitness as it was in section 2.1; since low target fitness could make every parameter sets find the target fitness easily in the most of the time.

In Figure 5.1 there are eleven parameter sensitivity 3D graphs with different elitism percentages. The x axis and y axis are the mutation rate and crossover rate. The z axis is the number of times that the best fitness found in the 50th generation is equal to the target fitness 1 in 1000 time trials with different parameter settings.

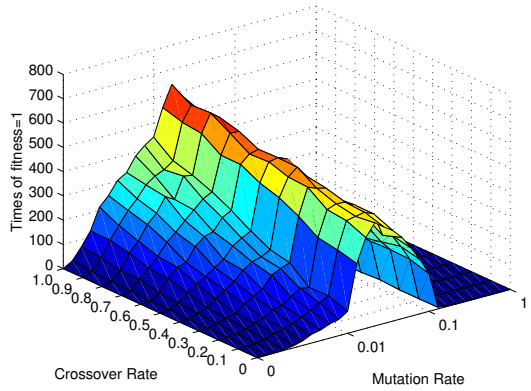
Figure 5.1 shows the amazing difference between elitism on and elitism off and the influence of the elitism percentage. Figure 5.1(a) shows that the genetic algorithm cannot find the best fitness at all if there is no elitism. When the elitism policy is 10%, In Figure 5.1(b), the genetic algorithm can find more than 600 times of the best fitness in 1000 trails. It means that the genetic algorithm has more than 60% percentage possibility to find the best fitness. It also indicated that the best mutation rate is about 0.01 and the crossover rate is about 0.9 which could mostly find the best result. In the following Figure 5.1(c), the possibility of finding the best fitness slightly increased when the elitism percentage was



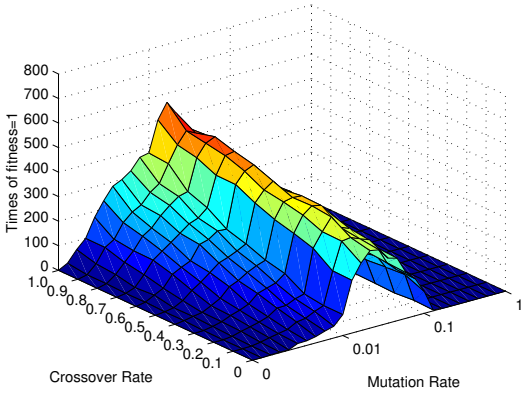
(a) 0%Elitism



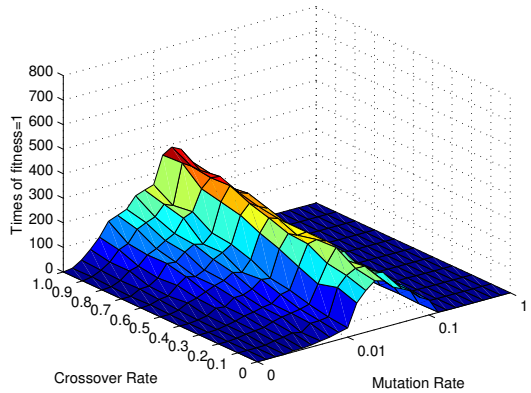
(b) 10%Elitism



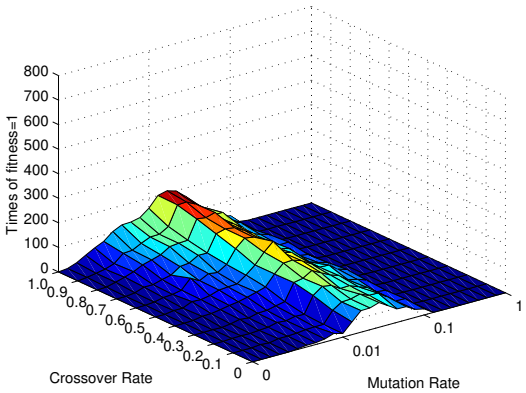
(c) 20%Elitism



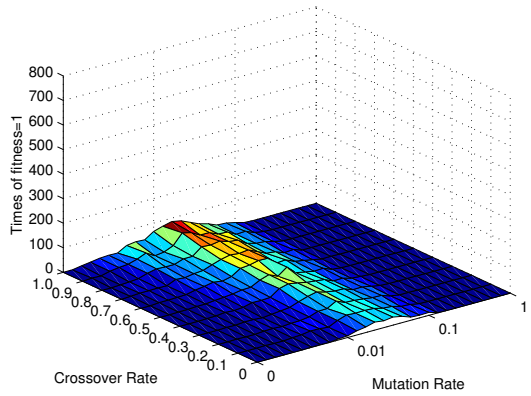
(d) 30%Elitism



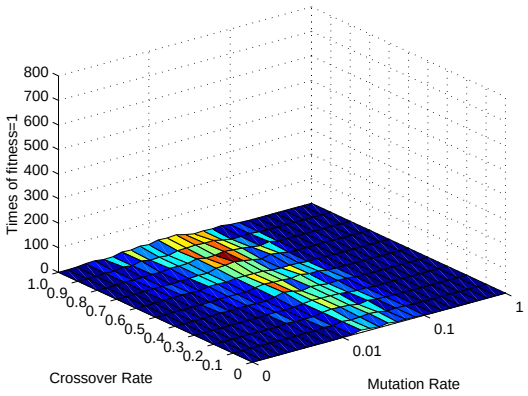
(e) 40%Elitism



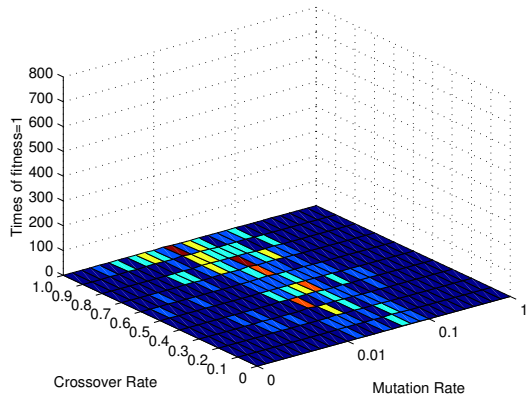
(f) 50%Elitism



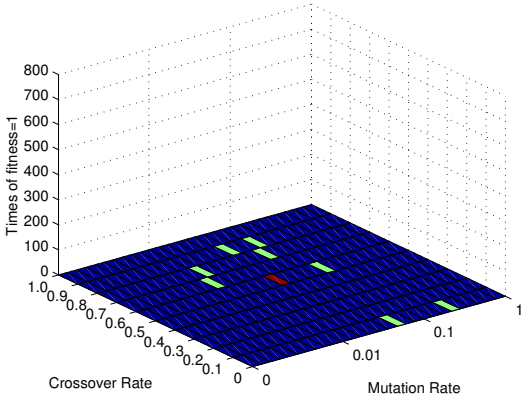
(g) 60%Elitism



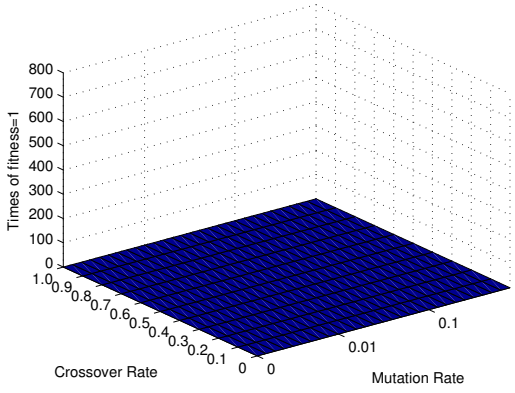
(h) 70%Elitism



(i) 80%Elitism



(j) 90%Elitism



(k) 100%Elitism

Figure 5.1: Elitism Percentage Testing

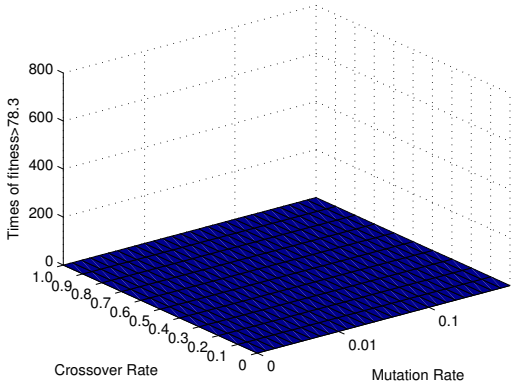
changed to 20%. However, the total number of times of finding the best fitness began to decrease when the elitism percentage reached to 30% in Figure 5.1(d). The total number of times of finding the best fitness quickly dropped down to below 400 times if the elitism percentage climbed to 40 %. According to Figure 5.1$(f)(g)(h)$ it is demonstrated that the general trend is for the total number of times of finding the best fitness being dramatically reduced with the increase of the elitism percentage. The last Figure 5.1$(i)(j)(k)$ reflects that it is hard to find the best fitness once the elitism percentage is larger than 80%. 100% elitism percentage means that no mutation and crossover happened from generation to generation. The original population was not changed and totally transferred to next generation. Therefore it is not possible to find best fitness.

In conclusion, the elitism plays a very important role in optimisation with the genetic algorithm. The result shows that a small elitism percentage helps new populations keep outstanding individuals from the old population and the population variety for new population. But the figures also show that the influence of elitism on finding the best fitness does not improve linearly with the increase of the elitism percentage as it changes from 0% to 100%.

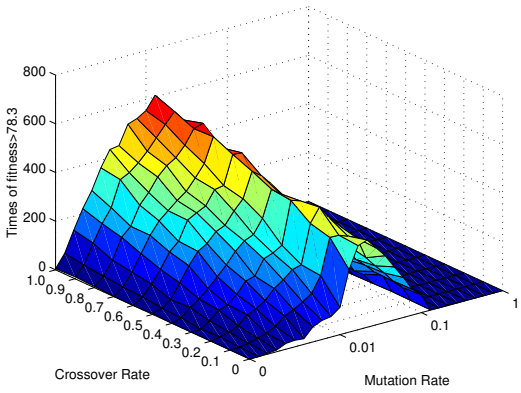
On the contrary, the optimal frequency decreases with the elitism percentage increasing when more than 30%. Too many elitisms lose the opportunity of creating new chromosomes, because the elitisms from the old population will be totally keep in the new generation. Figure 5.1 also shows that around 20% elitism percentage is a good choice for obtaining a quick and optimal result in SGA after comparing the optimal frequency of different elitism percentages.

5.1.2 De Jong's Function Testing

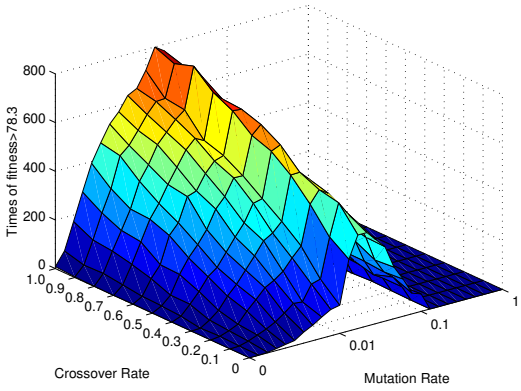
The function in Section 5.1 is a simple one which has one peak and smooth curve. In order to verify the correctness of the experiment result in the previous section, this section will choose one of De Jong's testing functions as a fitness function. Different from the function in the previous section, the De Jong's testing function selected represents a multi-



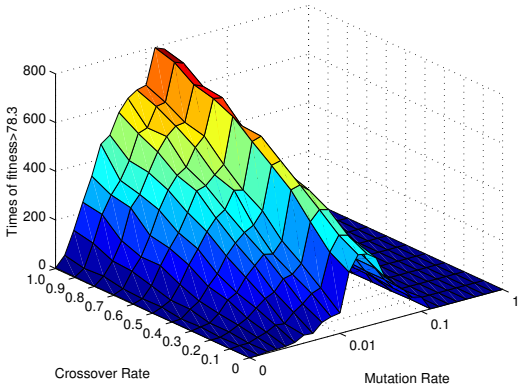
(a) 0%Elitism



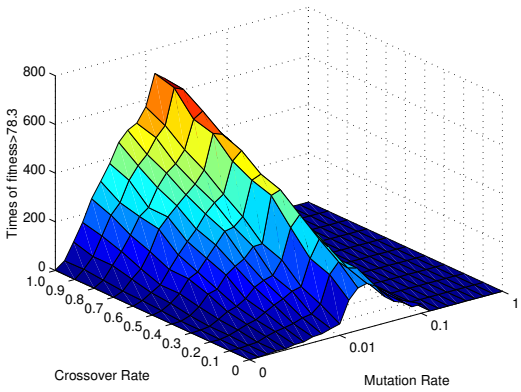
(b) 10%Elitism



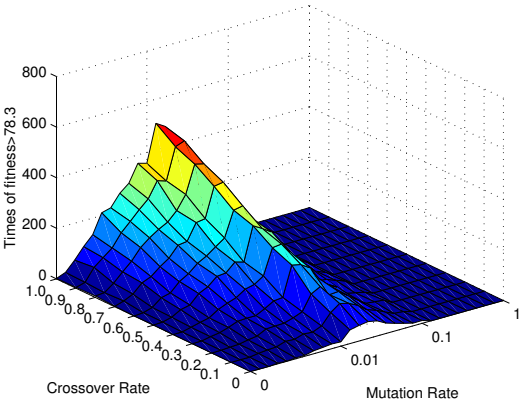
(c) 20%Elitism



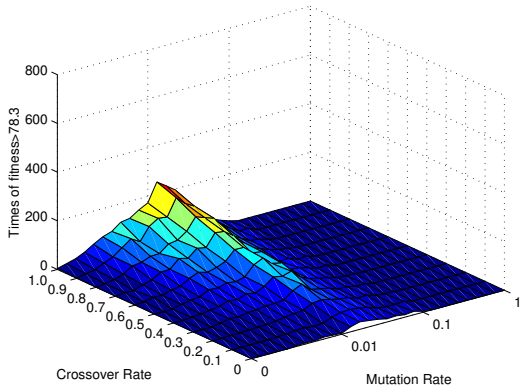
(d) 30%Elitism



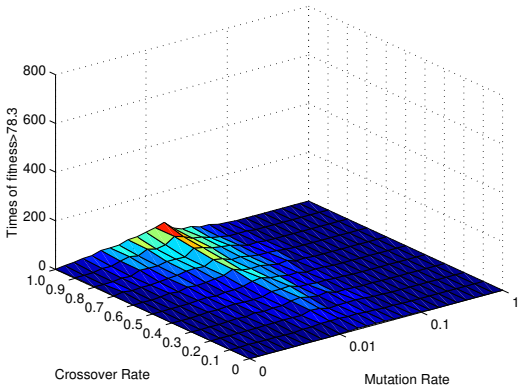
(e) 40%Elitism



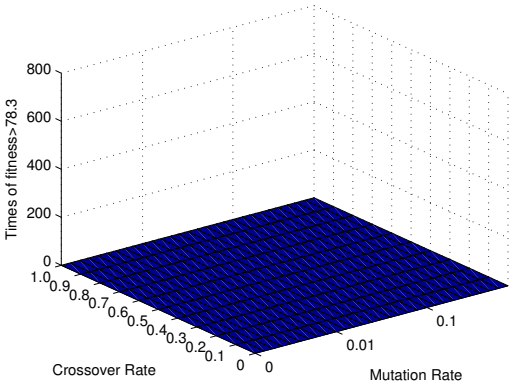
(f) 50%Elitism



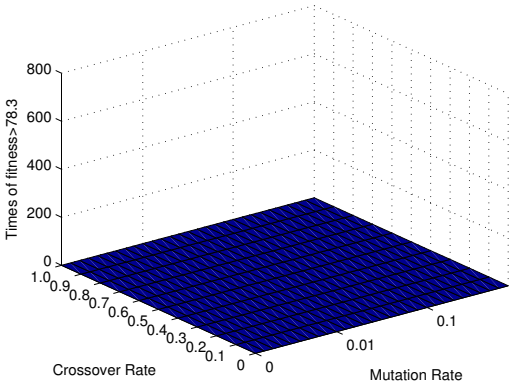
(g) 60%Elitism



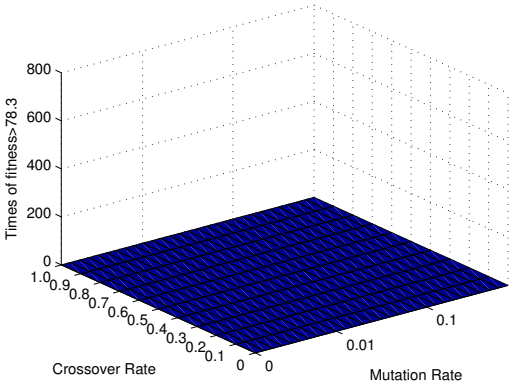
(h) 70%Elitism



(i) 80%Elitism



(j) 90%Elitism



(k) 100%Elitism

Figure 5.2: Elitism Percentage Testing with De Jong's Function

maximums function. The experiment in this section tries to explore whether the parameter sensitivity of GAs will change with the change of the fitness function.

$$f(x) = \sum_{i=1}^{3} x_i^2$$

Here x is the decimal value of the binary chromosome and $-5.12 \leq x_i \leq 5.12$. It has four maximum values in De Jongs testing function instead of one maximum value in the easy function mentioned in the previous section. The following experiment will redo the experiment in the previous subsection but with De Jong's function.

In Figure 5.2 there are eleven parameter sensitivity 3D graphs with different elitism percentage. The x axis and y axis are the mutation rate and crossover rate. The z axis is the number of times that the best fitness found in the 50th generation is equal to the target fitness of 78.3 in 1000 trials with different parameter settings. From Figure 5.2 it is suggested that the best crossover rate area is ranged from 0.7 to 1 and the best mutation rate is about 0.03.

As described above, the aim of this experiment is to explore the elitism influence for various types of optimisation function. Figure 5.2 demonstrated the changing of evaluation Renaults with the increasing of the elitism percentage. In Figure 5.1 the GA could find the best fitness=1 most times. However, it is hard to find the best fitness in the reasonable generations in Figure 5.2. It will be no change if the fitness is set too high or too low. The fitness which is observed to compare the quality of the evaluation is set to 78.3 by following the results in DeJong's thesis [25].

In Figure 5.2(a), it is hard to find the fitness which is larger than 78.3 without the elitism policy occurring in selections. As in Figure 5.1, Figure 5.2(b) shows the maximum time to find the observation fitness dramatically increase to nearly 600 when the elitism was involved in the selection. With the elitism percentage growing to 20%, the maximum time to find the asseveration fitness stably climbed to 800 in Figure 5.2(c). However, the maximum time to find the observation fitness barely changed by comparing the Figure

5.2(c) with the Figure 5.2(d). Figure 5.2(e) illustrates that the possibility, which could find the fitness more than 78.3, began to decrease when the elitism percentage rose to 40%. It clearly describes an obvious descent on the possibility of finding the observation of fitness when the elitism percentage has ascended to 50% in Figure 5.2(f). Figure 5.2(g) indicated a significant drop in the possibility while the elitism percentage was 60%. When the elitism percentage reached to 70%, the maximum time to find the observation fitness astonishingly fell below 50 in Figure 5.2(h). Figure 5.2(i), (j) and (k) depicts that there is no possibility in finding any fitness better than the observation fitness when the elitism percentage was larger than 80%.

Figure 5.1 and 5.2 provide some useful data regarding the influence of the elitism percentage in the selection. Although the optimal crossover rate area in Figure 5.2 is narrower than the one in Figure 5.1, the results are similar. The GA can find the best fitness a few times when there is no elitism. But the GA can find the best fitness most of the time when the elitism is 10%. And the number of times of finding the best fitness decrease when the elitism percentage is more than 40%. And Figure 5.2 suggests that the best elitism percentage in this function's optimisation is around 20% and 30%.

The experiment result shows that proper amount of elitism can keep the percentage of good fitness individuals in each generation, but too much elitism will destroy the diversity of individuals in each generation.

5.2 Sexual Selection Strategy

The elitism replacement policy directly affects the selective pressure but lacks the ability to control the genetic diversity. The sexual genetic selection, also called gender-specific selection, is a selection strategy which could balance the selective pressure and the genetic diversity at the same time.

In nature, male individuals try to spread their gene information as widely as possible and female individuals try to select the fittest males to mate with [108]. Inspired by the

natural behaviour of male vigour and female choice, sexual selection strategy [87, 107] applies two different selection mechanisms: male group (competitive) and female group (co-operative). The two different selection mechanisms provide the chance to balance the selective pressure and the genetic diversity. The winners (elitism) in the male group, which were picked up for the next generation with competitive mechanism, could maintain the selective pressure. The equal mating possibility in female groups with cooperative mechanism keeps the genetic diversity.

As mentioned, the running time of the fitness evaluation is considerable for automated tuning, because the fitness of each chromosome is the running cost to find the solution(s) with relative parameter set. In the standard genetic algorithm, there is only one selection strategy to choose the mating parents. It means that all the chromosomes involved the fitness evaluation. However the sexual selection strategy implements two selection mechanisms. Only the chromosomes in the male group involve the competition with their fitness. The female chromosomes don't need the competition and have the same opportunity for mating. It means that half of the fitness evaluation time was saved and the variety of the population was maintained. This is the most important reason why a sexual genetic algorithm was selected for the experiment.

Algorithm 3 Sexual Selection Strategy

1: Randomly generate the starting population P_i ▷ i is the population size
2: **for** $j = 1$ to n **do** ▷ j is the generation
3: **repeat**
4: Randomly select $i/2$ chromosomes of population as male
5: The rest $i/2$ chromosomes is marked as female
6: Evaluate the fitness of those male chromosomes
7: Select k elitisms from male chromosomes to mating pool
8: Each female chromosomes has the same possibility for mating
9: New generation is generated by mating k elitisms in male group and female chromosomes
10: **until** The best chromosome(solution) found or the running cost is out of time limit
11: **end for**

The pseudocode (Algorithm 3) of the sexual selection strategy clearly illustrates its working principle. Before the sexual selection strategy, the encoding, which transfers

the solutions of the optimisation problem to the chromosomes, is the first step as in standard genetic algorithm. After initializing the starting population, the population is divided into to two groups: male (competitive) and female (cooperative). The gender of each chromosome is randomly marked. The fitness of those male chromosomes was evaluated to select k elitisms for mating. Those k elitisms do the crossover and mutation with the randomly picked female chromosomes to generate the offspring until the new generation size is the same as the old generation. The generation loop repeats until the best chromosome(solution) found or the running cost reached the time limit.

5.3 Parallel Mechanism in Genetic Algorithms

From the pseudocode and the working principle of the sexual selection strategy, it indicated that the balance mechanism for the selective pressure and genetic diversity is feasible from the theory side. To verify the performance of the sexual selection strategy, the sexual genetic algorithm which implements the sexual selection strategy will compare with the standard genetic algorithm in tuning two solvers Minion and SAPS. Before the comparison, a parallel mechanism is introduced and implemented to the sexual genetic algorithm.

As mentioned in chapter 4, the main strategy of genetic algorithms for tuning is that each chromosome (parameter set) races against others. One of the most time consuming part for tuning is that elevate those parameter sets. Therefore it deserves further study to reduce those time cost. Parallel mechanism in genetic algorithms [3] is an ideal approach to solve this task. Due to the implicit parallelism in genetic algorithms [16], many variant parallel strategies were posed and applied. For example the population were divide into several small populations and they will do the evolution simultaneously. However the main idea implemented in this thesis is that the parameter sets in each generation will be evaluated simultaneously. To justify the efficiency of parallel mechanism, a parallel genetic algorithm (PGA), which implements such parallel mechanism to GACM, will compare with the GACM by tuning the same twenty instances as in section 4.4.3. Commonly the

Instance	Cutoff(s)	PGA Running time(ms)	GACM Running time(ms)
N-Queen(26)	1	3224	15858
N-Queen(27)	1	3242	18008
N-Queen(28)	4	11980	52831
N-Queen(29)	2	5922	31051
N-Queen(30)	1	160467	895120
BIBD1	1	341	1069
BIBD2	1	567	1467
BIBD3	1	3233	11649
BIBD4	1	2978	10686
BIBD5	1	2673	13098
Golomb(6)	1	2764	5857
Golomb(7)	1	2960	11640
Golomb(8)	1	3278	20372
Golomb(9)	5	15321	135604
Golomb(10)	35	105327	1017300
Langford(2,10)	1	3310	22634
Langford(2,19)	6	17517	78126
Langford(2,20)	5	14482	63822
Langford(3,17)	4	12217	71353
Langford(3,19)	4	12207	90467

Table 5.1: The Comparison Between Parallel Genetic Algorithm and GACM

amount of parallel tasks are restricted by the CPU computation ability. Thus the number of the parallel tasks that evaluate the quantity of each parameter sets is set as ten.

The difference between the parallel genetic algorithm and GACM in this section is with or without parallel mechanism. It means that the final parameter sets found by two approaches are the same or very close. Thus the experiment only focuses on the running cost for the tuning. The comparison between the parallel genetic algorithm and GACM is to compare the average tuning cost for each instance over 10 times repeat trails.

Table 5.1 shows the running cost difference between the PGA and GACM for tuning. There are four columns in the table. The first column is the name of those twenty instances. The second column is the cut off time for evaluating each parameter set. The third and fourth columns are the running cost of tuning with PGA and GACM. It clearly shows that the running cost was greatly reduced by implementing the parallel mechanism.

It shows that the running cost for tuning those twenty instances is reduced in different ranges. Although the number of the parallel tasks is ten, there is only one running cost for tuning reduced to nearly one tenth. To compare with the whole tuning strategy, the evaluation time for each parameter set does not occupy most of the CPU time is the main reason. In another word, the evaluation time does not hold very large proportion in most instances chosen. When the evaluation time of parameter sets increase such as Golomb (10), the improvement is obvious.

5.4 Sexual Genetic Algorithm for Tuning

In the previous sections, three strategies which could lead to the performance improvement are mentioned: the elitism selection policy, the sexual selection strategy and the genetic parallel mechanism. In this section, a new sexual genetic algorithm, which combines the elitism selection policy and the parallel mechanism, is implemented for tuning. To justify the performance of the sexual genetic algorithm, it is compared with the gender genetic algorithm [5] which is a state-of-the-art generic tuning algorithm.

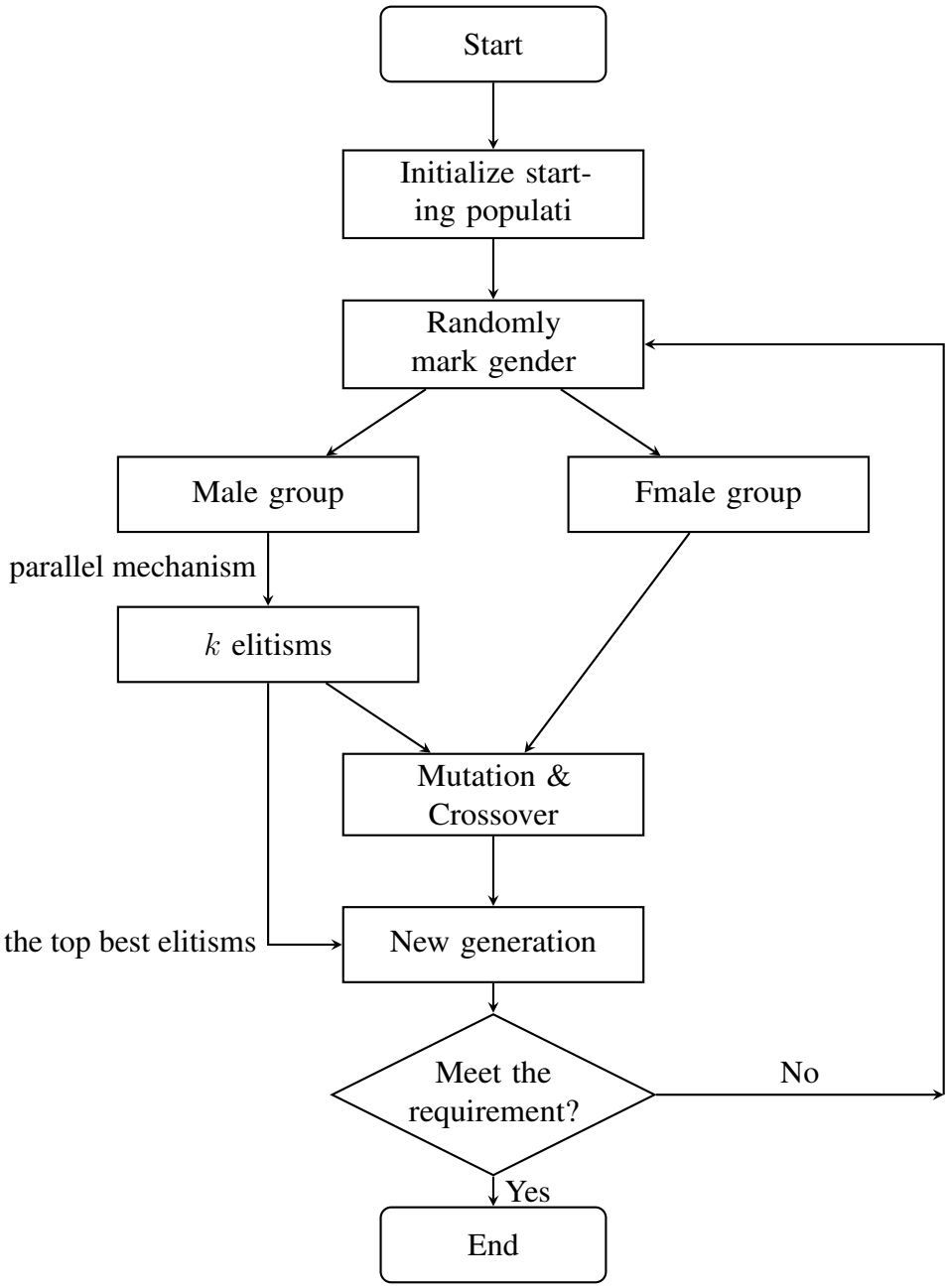


Figure 5.3: The Flowchart of Sexual Genetic Algorithm

Compared with gender genetic algorithm, the elitism strategy mentioned will be implemented in the sexual genetic algorithm. The encoding in the gender genetic algorithm is real value coding which is easy to understand. But the argument between the real value coding and binary coding has never stopped. However, the binary coding is similar to the machine coding and it always shows it is superior when it was applied to deal with categorical values [1]. This thesis implements the binary code as the encoding way.

Figure 5.4 indicates the flowchart of the sexual genetic algorithm. Compare with the sexual selection strategy in section 5.2, the parallel mechanism is implemented in the male competition to gain the k elitisms in a quicker way. Meanwhile, the top best individual(s) from the k elitisms are kept in the new generation by the elitism selection policy. In the gender genetic algorithm, the parameter sets were divided into subgroups and each subgroup was regard as a thread in the parallel mechanism. The aim is to find the k elitisms quickly. However in the sexual genetic algorithm, all the parameter sets were evaluated with parallel mechanism in one group with the same amount threads as in gender genetic algorithm. Once any parameter set finishes the evaluation in any thread, it will be replaced by another waiting parameter set. The threads stop until the k elitisms found.

5.4.1 Sexual Genetic Algorithm VS. Gender Genetic Algorithm in Tuning Minion

To justify the efficiency, sexual genetic algorithm and gender genetic algorithm are firstly implemented in tuning Minion to compare the performance. Since the gender genetic algorithm is open source, the source code of GGA is obtained from website and is directly implemented in the comparison experiment of SGA versus GGA. The gender genetic algorithm and the sexual genetic algorithm were implemented to reproduce the experiment in section 5.3 instead of parallel genetic algorithm. In the experiment, the amount of the parallel tasks is set to eight which is the same as in [5]. Following the setting in the gender genetic algorithm's paper, the gender genetic algorithm and the genetic algorithm attempt

Instance	SGA(ms)	GGA(ms)	Instance	SGA(ms)	GGA(ms)
N-Queen(26)	33	169	Golomb(6)	1	2.6
N-Queen(27)	29	34	Golomb(7)	20	289
N-Queen(28)	32	64	Golomb(8)	215	304
N-Queen(29)	32	32	Golomb(9)	1772	3169
N-Queen(30)	37	69	Golomb(10)	1737	2727
BIBD1	28	29	Langford(2,10)	199	256
BIBD2	25	29	Langford(2,19)	32	33
BIBD3	32	46	Langford(2,20)	38	39
BIBD4	32	43	Langford(3,17)	162	360
BIBD5	35	38	Langford(3,19)	215	284

Table 5.2: Sexual Genetic Algorithm

to find the top 20% of the elitisms from male group for mating. Since the sexual genetic algorithm implements the elitism selection strategy, the sexual genetic algorithm always keeps the best elitism from the old generation to the new generation. The performance of the sexual genetic algorithm compares with gender genetic algorithm in tuning minion over twenty instances. We will repeat the trials ten times and observer the average.

Table 5.2 illustrates their efficiency over twenty instances. The left side column in the table is the relative instance name. The data in the columns is the minimum running of the best parameter sets found by those two algorithms. It shows the superiority of the sexual genetic algorithm over gender genetic algorithm. Even the worse performance of the sexual genetic algorithm in N-queen(29), the minimum found by the sexual genetic algorithm is equal to what the GGA did.

5.4.2 Sexual Genetic Algorithm VS. Gender Genetic Algorithm in Tuning SAPS

In [5] the gender genetic algorithm demonstrated its efficiency by tuning Scaling and Probabilistic Smoothing (SAPS), which is a high-performance boolean satisfiability problem solver (algorithm). In that paper, the gender genetic algorithm was implemented to tune the SAPS in solving colour mapping problems. To verify the performance of the

	SGA		GGA	
Run (Instance 1006)	Minimum(ms)	Cost(ms)	Minimum(ms)	Cost(ms)
1	52	2371	52	5608
2	48	2733	64	5522
3	40	2653	52	5492
4	50	2821	54	4936
5	32	2878	67	5401
6	52	2921	46	4513
7	50	2589	46	5137
8	40	3169	50	4543
9	46	2792	43	4889
10	48	3022	46	5278

Table 5.3: The Comparison Between Sexual Genetic Algorithm and Gender Genetic Algorithm in Solving Instance 1006

	SGA		GGA	
Run (Instance 10013)	Minimum(ms)	Cost(ms)	Minimum(ms)	Cost(ms)
1	24	1740	36	2813
2	20	1866	16	2694
3	18	1644	23	2402
4	18	1851	21	3602
5	20	1690	30	1531
6	20	2023	33	2632
7	24	1753	29	2851
8	24	1727	18	2612
9	16	1580	29	2667
10	18	1596	29	2189

Table 5.4: The Comparison Between Sexual Genetic Algorithm and Gender Genetic Algorithm in Solving Instance 10013

	SGA		GGA	
Run (Instance 10017)	Minimum(ms)	Cost(ms)	Minimum(ms)	Cost(ms)
1	56	3119	41	5689
2	50	2632	53	5163
3	46	3028	56	6219
4	46	2810	52	6127
5	54	3040	46	5223
6	48	3459	34	4934
7	54	2915	61	5593
8	60	2904	56	4805
9	46	2798	47	6012
10	56	2736	41	6138

Table 5.5: The Comparison Between Sexual Genetic Algorithm and Gender Genetic Algorithm in Solving Instance 10017

	SGA		GGA	
Run (Instance 10055)	Minimum(ms)	Cost(ms)	Minimum(ms)	Cost(ms)
1	8	948	7	1142
2	8	938	20	860
3	8	983	16	751
4	8	905	9	1945
5	6	942	12	1250
6	8	984	13	1634
7	6	806	12	1378
8	6	935	20	2037
9	6	855	10	1509
10	8	965	7	803

Table 5.6: The Comparison Between Sexual Genetic Algorithm and Gender Genetic Algorithm in Solving Instance 10055

sexual genetic algorithm in further, twenty instances of the colour mapping problem were randomly selected from the benchmark instance in that paper. SAPS has four parameters for tuning: alpha, rho, ps and wp. According to the value of those four parameters, the chromosome length was set to 12. The population is set to 20, the generation is 10, and the mutation rate is 0.1 and crossover rate is 0.9. Other settings are the same as in last experiment.

To illustrate the efficiency of the sexual genetic algorithm, four instances' result were randomly selected to list in Table 5.3, 5.4, 5.5 and 5.6. The result demonstrates the performance of those two algorithms on four instances over ten times run. Table 5.7 is the statistical data based on four tables. In the table 5.7, it clearly illustrates the minimum running cost of parameter sets found and running cost of tuning by the sexual genetic algorithm and gender genetic algorithm. Although the tuning time cost of sexual genetic algorithm is less than the gender genetic algorithm need. From the mean of those four instances, it shows that the sexual genetic algorithm still could find a better parameter sets than the GGA. From the standard deviation in the table 5.7, it shows the sexual genetic algorithm is more stable and reliable. As mentioned, twenty instances were randomly selected from the benchmark data, and four instances to show the performance difference between SGA and GGA. Although other sixteen instances' result were not listed, the conclusion is the same as those four instances listed.

	SGA (mean ± stddev)		GGA (mean ± stddev)		
Instance	Minimum(ms)	Cost (ms)	Minimum(ms)	Cost (ms)	Table
1006	45.8 ± 6.16	2794.9 ± 214.73	52.0 ± 7.52	5131.9 ± 378.09	5.3
10013	20.2 ± 2.75	1747.0 ± 128.91	26.4 ± 6.23	2599.3 ± 497.77	5.4
10017	51.6 ± 4.80	2944.1 ± 222.36	48.7 ± 7.95	5590.3 ± 503.57	5.5
10055	7.2 ± 0.98	926.1 ± 53.83	12.6 ± 4.52	1330.9 ± 433.64	5.6

Table 5.7: Test performance Comparison(mean runtime over test instances, in CPU Milliseconds)

5.5 Conclusion

A 3D elitism graph of genetic algorithms clearly shows that the elitism is very important and amazing for general genetic algorithm optimisation. The elitism makes the best solution finding of genetic algorithm not only possible but also quickly in the easy function and the De Jong's Function Testing.

The experiment also suggests that about 20% elitism is a good percentage for genetic algorithms. A small elitism percentage can promise that the good individuals in an old generation will not be lost with the new generation creating. Too much elitism loses the opportunity of creating new chromosomes because the elitisms from the old population will occupy too many portions in the new generation. Too much elitism loses the diversity of the population since the chromosomes in the population have a high fitness that easily leads to a local optimal trap.

In the experiment of the parallel mechanism, it shows that the parallel mechanism great improves the efficiency of the sexual genetic algorithm. The sexual genetic algorithm combined with proper elitism selection may achieve a better efficiency due to a suitable selective pressure. However, the search will slow down with an incorrect percentage elitism selection because of a lack of genetic diversity. The results in the experiment suggest that a sexual genetic algorithm with low percentage elitism is practicable. The tuning results for Minion and SAPS show that the sexual genetic algorithm is an efficient and stable method

for tuning.

Chapter 6

Self-learning Genetic Algorithm

GACM and sexual genetic algorithm are two algorithms that were proposed in previous two chapters to help tuning based on a single instance. The instance-based tuning [5] is another type of tuning approach, which has a given collection of train instances. The aim of instance-based tuning is to seek an optimal parameter set from the train instances for the testing instances. The following two chapters attempt to explore genetic based algorithms for instance-based tuning.

Machine learning [4, 10] is one of the most important and indispensable branches of artificial intelligence. The aim of machine learning is to extract the useful information or knowledge from the existing data or previous results [19, 91]. That knowledge or experience will help to speed the searching or optimising for the similar problems [73].

In this chapter, a genetic based self-learning strategy is proposed for the instance-based tuning. The first section introduces some relevant approaches in machine learning and to justify the reason to choose genetic algorithms for the instance-based tuning. Next, the self-learning genetic algorithm which implements the iteration idea in ParamILS is introduced. To justify its efficiency, some experiments are designed in the Section 6.4. The performance of the self-learning genetic algorithm was compared with ParamILS in tuning Minion and SAPS. Finally the experiment's results are concluded in the closing section of the chapter.

6.1 Preliminaries

6.1.1 Machine Learning

Arthur Samuel first gave an informal definition of machine learning in his paper in 1959. It said: ”Machine learning is a field of study that gives computers the ability to learn without being explicitly programmed” [95]. Later, Mitchell posted a more formal and rigorous definition for machine learning: ”A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E”[77]. In essence, machine learning could also be understood as a kind of approximation for the exact model of real problems. The aim of machine learning is to build up a modelling with the training data and to reduce the error between the model generated and the exactly model of real problems.

The following will introduce some of the most popular machine learning strategies: K-means clustering, neural networks and support vector machines.

6.1.2 K-means Clustering

One of the most straightforward clustering algorithms is Lloyd’s K-means [71]. In K-means Clustering firstly selects k random points in the feature space. It then alternates between two steps until some termination criterion is reached. The first step assigns each instance to a cluster according to the shortest distance to one of the k points that were chosen. The next step then updates the k points to the centres of the current clusters. While this clustering approach is very intuitive and easy to implement, the problem with k-means clustering is that it requires the user to specify the number of clusters k explicitly. If k is too low, this means that some of the potential is lost to tune parameters more precisely for different parts of the instance feature space. On the other hand, if there are too many clusters, the robustness and generality of the parameter sets that are optimised for these clusters is sacrificed. Furthermore, for most training sets, it is unreasonable to assume that the value of k is known.

6.1.3 Neural Networks

In general, neural networks with one hidden layer, a non-linear activation function and a sufficient number of hidden neurons are able to approximate any function with arbitrary precision. However, the error function is not convex and thus the result of the training depends on the initialisation.

There are several steps which are important if one uses a neural network: One has to check that no other more efficient method is available and that the problem can be treated using an artificial neural network (feasibility). One has to plan the project. This includes resources, personnel, costs, and documentation. The next step is to setup standards for data collection and coding. After data is collected, one has to assure the quality of the data. Now the network can be designed. Usually one needs several training cycles to obtain an optimal structure of the network. After the network has been trained and designed, one has obtained precise estimates for the errors. One can use the trained network to extract rules. Rules are important for optimisation and control of the network.

However, it would require lots of computational resources to fully implement a standard neural network architecture. Neural networks require a large amount training sets to be trained properly and to give output(s) that would be close enough to the desired output but knowing what amount of training sets, is enough for a desired output would be totally dependent on the trainer itself - but yes it's important that a very large training set is provided so that the neural network would have sufficient understanding of the underlying structure.

6.1.4 Support Vector Machines

SVM is a supervised machine learning algorithm which can be used for classification or regression problems. It uses a technique called the kernel trick to transform your data and then based on these transformations it finds an optimal boundary between the possible outputs. Simply put, it does some extremely complex data transformations, and then

figures out how to separate your data based on the labels or outputs you've defined.

One of SVM drawback is that the complex data transformations and resulting boundary plane are very difficult to interpret. Another disadvantage of the SVM algorithm is that it has several key parameters that need to be set correctly to achieve the best classification results for any given problem.

The above description listed the drawbacks and advantages of the recent popular machine learning strategies. There is more or less difficulty on implementing them. The previous chapters have shown the efficiency and feasibility of genetic algorithms in tuning. This chapter will focus how to adapt the genetic algorithms to learn from the training sets. As with machine learning strategies in tuning, the aim of the training is to find an optimal parameter set which has the best average performance for all the training sets.

6.2 Self-learning Genetic Algorithm

The self-learning genetic algorithm (referred to as SLGA) is an instance based tuning algorithm which make the prediction by extracting experience on the sample instance. Self-learning, which learns its own inductive bias based on previous experience, is one of the typical algorithms in the machine learning domain. Self-learning could avoid repetition of searching and computation in the previous experiments.

The last few chapters explored various ways to improve the search ability of a genetic algorithm by creating different strategies such as balancing the selective pressures and the genetic variety. In standard genetic algorithm, the starting population is randomly generated because the search domain is unknown and the random chromosomes keep the variety of the population to prevent early convergence in evaluation. But it is mentioned in chapter 2 that the quality of the starting population is a considerable factor, as with the crossover rate and the mutation rate. The searching ability of genetic algorithm can be improved by narrowing the starting population domain [40].

Combining with the influence of the starting population in genetic algorithms and the

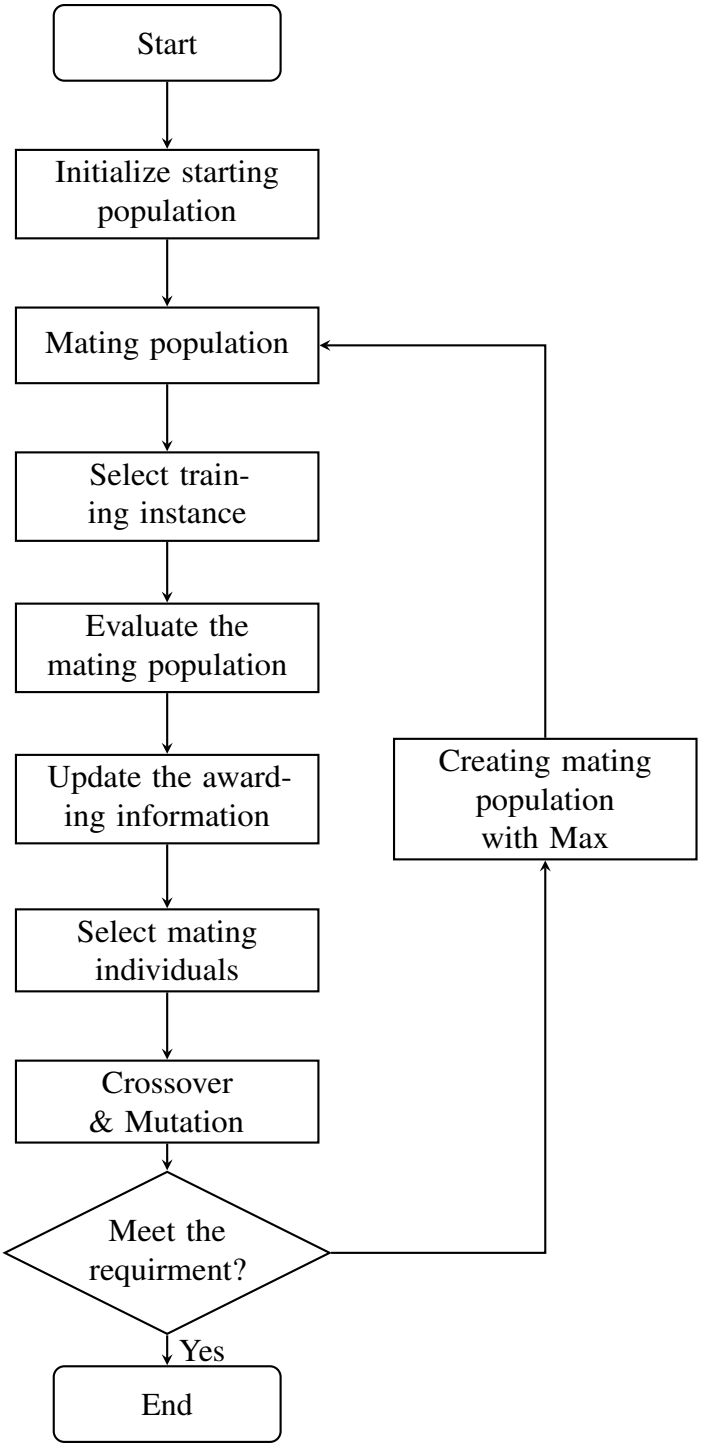


Figure 6.1: The Flowchart of Self-learning Genetic Algorithm

idea of self-learning, the main idea of the self-learning genetic algorithm was formed. To learn from the training sets, a random sample instance of all available training instances is selected in each generation. The performance of the whole population is evaluated and the individuals race against each other. After mating, the new population is passed to the next generation as normal. Therefore the subsequent generation inherited all the good parameter sets for the last sample instance.

As mentioned in the introduction, ParamILS is an iterated local search or statistics based tuning algorithm. ParamILS implements a special iteration technique to limit the number of training instances that need to be run for each parameter set. It starts with a random assignment of all the parameters. Meanwhile it gathers statistics on which parameters are important. The current best parameter set would be replaced only if a new parameter set has been evaluated on at least as many training instances as the current best. A similar statistical approach called bonus strategy will mentioned and implemented in our self-learning genetic algorithm.

Figure 6.2 clearly demonstrates the flowchart of self-learning genetic algorithm. Actually it is implemented in the following way:

Initialisation - A few populations were firstly initialized such as the starting population P for the evolutionary, the mating population PM and the best population PB, which gather the best parameter set in each generation. At begin the mating population is equal the starting population, else it is the total of the population P and the best population PB. Meanwhile some other variables were initialized as well. According to the value of the tuning parameters, the chromosome length $choromlength$ will be initialised. $BestParam$ is the variable to store the best parameter set in each generation.

Bonus Strategy - As in the ParamILS, the self-learning genetic algorithm implements an iteration approach called bonus strategy to record each best parameter set in each generation. Therefore array B_k is initialized to record the occurrence frequency of each best parameter happened in each generation, where k is the size of arrayB_k. The default

value of k is the size of the generation. However, when k is less than the generation size, the recent best parameter set will replace the worst or the early on in B_k. When the self-learning genetic algorithm finishes, the bonus mechanism will find out an optimal parameter set which has highest occurrence frequency or is the best parameter in the last generation.

Mating Rule - In self-learning genetic algorithm, a random instance T will be chosen from the training sets for each generation. To help each parameter set has the chance for mating, the selection strategy in here is the roulette wheel.

Algorithm 4 Self-Learning Genetic Algorithm

```
1: Initialize P, PB, PM, B_k
2: PM ← P
3: for i ← 1 to n do
4:     T ← ChooseInstance(TrainingSets)
5:     BestParam ← Fitness(PM)
6:     if BestParam == B_k then
7:         B_k ← B_k + 1
8:     else
9:         B_i ← best configuration
10:    end if
11:    Parents ← Select(PM)
12:    for j ← 1 to n do
13:        P ← Crossover(Mutation(Parents))
14:    end for
15:    if Checktime()==true then
16:        break
17:    else
18:        PM ← P ∪ PB
19:    end if
20: end for
21: if Bouns()==true then
22:    λ ← B_best
23: else
24:    λ ← BestParam
25: end if
26: return λ
```

Crossover and Mutation - In the self-learning genetic algorithm the single point crossover is applied. The mutations which change one or more genes in an individual is another operator used in GA. The self-learning genetic algorithm implements the most classic one point mutation.

The pseudocode of the self-learning genetic algorithm illustrated its working principle and the way of realizing the idea to the programming code.

6.3 The Performance of the Self-learning Genetic Algorithm

After the introduction of self-learning genetic algorithm, its performance is firstly verified by tuning Minion in solving various CSPs. The optimisation problems involved in the testing are the BIBD, the N-queen problem, Golomb, and the Langford's Number problem. In the performance testing, the self-learning genetic algorithm attempts to find an optimal or best parameter settings for the large instance by training the small instances of the same CSPs.

Since the bonus strategy of the self-learning genetic algorithm is a kind of statistical approach, its performance will compare with ParamILS in tuning another solver SAPS to justify the efficiency as in chapter 5.

6.3.1 The Distribution of Parameter Sets

Before the performance testing, the distribution of the running time with different parameter sets on four optimisation problems will be investigated. The aim of the investigation is to discover whether the parameter sets distribution change with different instance for the same type problem. Therefore each optimisation problem will select three instances for exploring the distribution. In the experiment all the parameter sets in each instance will be evaluated and noted.

Figure 6.2 and Figure 6.3 illustrate that the running cost distribution of the possible parameter set in solving different instance. X axis in these graphs is the running time of

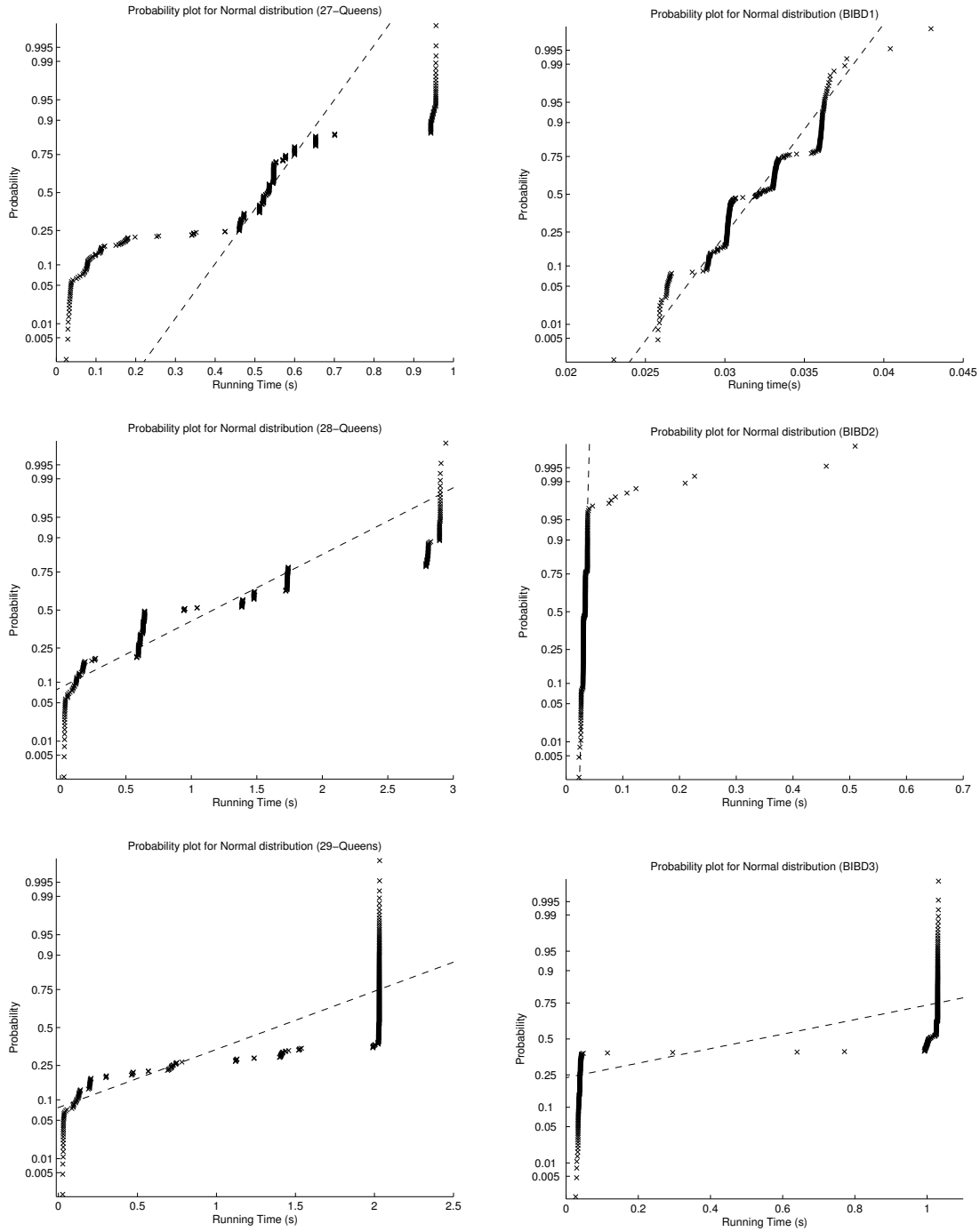


Figure 6.2: The Distribution for N-queen and BIBD

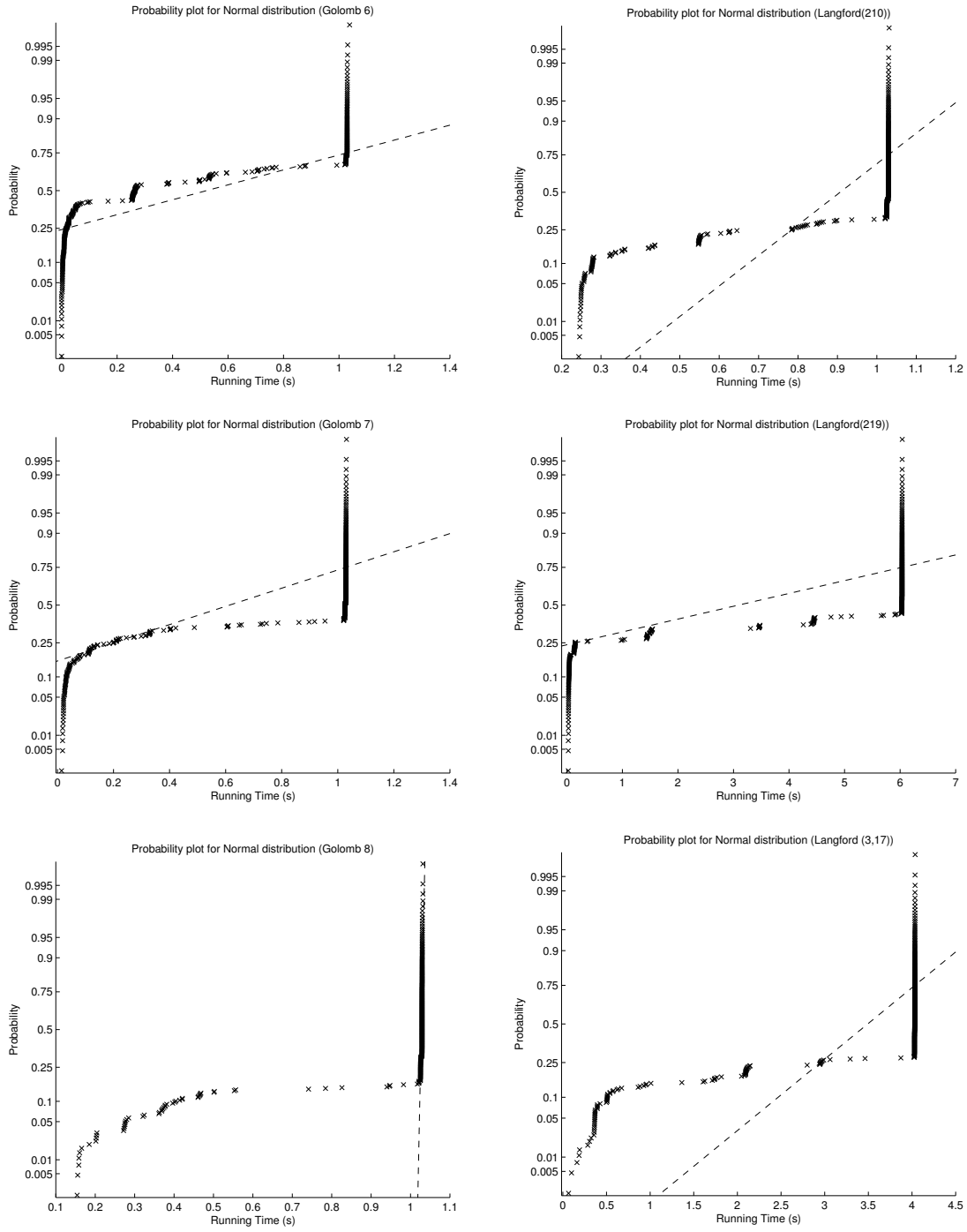


Figure 6.3: The Distribution for Golomb Ruler and Langford Number Problem

specific parameter set. Y axis is the probability that parameter sets occupied. When the value of Y is 1, it presents all parameter sets. From the result, it shows that the distribution is more or less changed by the instances changing in the same problem. In another word, the optimal parameter set changes in different instances. Those two figures also indicate that the changing with different instance in n-queen, Golomb Ruler and Langford Number problem is not dramatical. To compare with other problems, the changing in BIBD is more obvious.

6.3.2 The Performance Comparison in Tuning Minion

To justify the performance in tuning Minion, the self-learning genetic algorithm was implemented to tune Minion. In the comparison there will be three training instances and one testing instance for each CSPs. The population size will be ten, crossover rate is 0.9 and mutation rate is 0.1. Therefore the self-learning genetic algorithm will implement one second as the cutoff time for each training instance. The number of parallel task for self-learning genetic algorithm is ten. The k in bonus strategy is the same as population size. Each trial was run ten times and I observed the average of the minimums.

Table 6.1 shows that the performance of the self-learning genetic algorithm in tuning Minion. The left two columns is the name of three training instances and one test instance. The third column is the total training time for those three test instances. The fourth column is the solving time for the test instance with the candidate parameter sets found by small training instances. The ”total” column is the total time which includes the training time and solving time. The right column is the default running time for running the large instance with the default parameter setting. Although the performances are different, the result of the performance in Table 6.1 is obvious in showing that the self-learning genetic algorithm is feasible for finding a better parameter sets for large instances. The running time for find the solutions for 100-queens is over 240 times faster than the one with default.

Train Instance	Test Instance	Training Time	Solving Time	Total	Default
26-Queens					
27-Queens					
28-Queens	30-Queens	30s	0.07s	30.07s	87s
26-Queens					
27-Queens					
28-Queens	100-Queens	300s	0.2s	300.2s	>20hrs
BIBD1					
BIBD2	BIBD				
BIBD3	(9,90,40,4,15)	300s	2911s	3211s	3674s
Golomb (6)					
Golomb (7)					
Golomb (8)	Golomb(10)	10s	30s	40s	62s
Golomb (6)					
Golomb (7)					
Golomb (8)	Golomb(11)	300s	4169s	4469s	9290s
Langford(2,10)					
Langford(3,17)					
Langford(3,19)	Langford(2,19)	3s	0.29s	3.29s	6s
Langford(2,10)					
Langford(3,17)					
Langford(3,19)	Langford(2,27)	300s	1223s	1523s	>12hrs

Table 6.1: The Performance of SLGA in Tuning Minion

6.3.3 The Performance Comparison in Tuning SAPS

In [55] ParamILS shows its efficiency in tuning SAPS with its two tuning algorithms Basic and Focus local search. As mentioned, the statistical idea in the self-learning genetic algorithm for the optimal parameters found in each generations is derived from ParamILS. Therefore ParamILS is closely related to the proposed SLGA.

To verify the influence made by the import of the genetic algorithm and its parallel mechanism, the self-learning genetic algorithm will be implemented to tune SAPS with the same instances in the ParamILS's paper. Since ParamILS is an open source project for the research purpose, its source code is directly obtained from the internet. ParamILS will reproduce the experiment in the paper with its source code in the same platform as the self-learning genetic algorithm does.

In the experiment, the benchmark set for SAPS consists of 113 SAT instances for training and 100 different instances for testing. For SAPS, I ran self-learning genetic algorithm for 200 generations with a population of size 60. The crossover rate will be 0.9 and mutation rate will be 0.1. The total CPU time for tuning is 1800 seconds which was used for self-leaning genetic algorithm and ParamILS when configuring SAPS. The cutoff time for evaluating each parameter set in each training instance for both tuning approaches is one second. In the comparison, each tuning algorithm will run twenty runs.

Table 6.2 illustrated the final average computation time for the instances in the training as well as the test set. Although the Basic ParamILS is more stable than Focus ParamILS, the 20 tuning runs shows that Basic ParamILS is not good as Focus ParamILS. It is obvious that the performance of self-learning genetic algorithm outperforms both basic ParamILS and focus ParamILS. Self-learning genetic algorithm's worst parameter set results in an average performance of 27 ms per test instance, which is still better than the best Focus ParamILS parameter set which requires 68 ms. Self-learning genetic algorithm also showed it is superior on the performance of test instance. The standard deviation shows that the self-learning genetic algorithm is more stable than other two algorithms as well. The results of the comparison with ParamILS clearly shows the efficiency and the improvement

	Basic ParamILS		Focus ParamILS		SLGA	
Run	Train(ms)	Test(ms)	Train(ms)	Test(ms)	Train(ms)	Test(ms)
1	316	280	84	94	18	17
2	315	287	71	73	19	19
3	316	283	75	67	17	13
4	317	285	72	71	23	12
5	316	282	68	73	16	11
6	317	278	73	64	14	14
7	316	277	86	99	21	22
8	315	277	68	73	19	23
9	317	280	71	67	23	18
10	316	281	69	63	22	21
11	315	274	73	76	10	13
12	317	271	72	76	25	27
13	317	278	70	78	27	24
14	318	275	73	65	25	25
15	315	278	73	66	22	26
16	317	282	76	66	19	19
17	317	282	68	78	13	10
18	315	281	76	71	15	13
19	315	280	72	71	18	16
20	316	277	69	69	22	21
Mean	316.2	279.4	73.0	73.0	19.4	18.2
STDDEV	0.91	3.7	4.7	9.0	4.3	5.2

Table 6.2: The Efficiency of Self-Learning Genetic Algorithm in Tuning SAPS by comparing ParamILS

of using SLGA over ParamILS on tuning SAPS, when the iteration local search idea in ParamILS was combined with the genetic algorithm.

6.4 Conclusion

Actually, the aim of this chapter is to propose and justify an instanced-based tuning approach called self-learning genetic algorithm, which could learning the experience from other training instances in the same problem. The self-learning genetic algorithm is proposed by combing the ideas on the starting population and iteration.

The comparative study of the evolutionary speed between different starting populations indicated that the quality of the starting population could affect the evolutionary speed and results. In this chapter the generation of each instance will be regarded as the starting population for other instances. The population will be updated with the involving of the new random training instance in each generation. Therefore the whole population do the evolutionary towards the direction that finds an optimal parameter set for the average performance of training instances.

Meanwhile the iteration idea in ParamILS was implemented and revised as the Bonus Strategy in the self-learning genetic algorithm. This mechanism applied a statistical approach to recode the best parameter sets in each generation and their occurrence frequency. This idea helped the self-learning genetic algorithm to locate the best parameter set in further. In practice, our self-learning genetic algorithm has demonstrated it is superior over ParamILS by comparing in the Minion tuning and SAPS tuning.

Chapter 7

Self-learning Sexual Genetic Algorithm

The main focus of the thesis is to develop automatic tuning algorithms for CSPs solver. For the single instance tuning, GACM and a sexual genetic algorithm were proposed in this paper. For the instance-based tuning, the standard genetic algorithm was modified and improved as a new self-learning genetic algorithm in the previous chapters. Although the sexual genetic algorithm and the self-learning genetic algorithm are both based on the standard genetic algorithm, they implemented different strategies to tune different type of tuning. The hybrid strategy is expected to be explored for instance-based tuning in this chapter.

The self-learning sexual genetic algorithm will be introduced in this chapter. The first section will introduce the principle and the structure of the self-learning sexual genetic algorithm. Section 7.2 justifies the efficiency of the self-learning sexual genetic algorithm by comparing with self-learning genetic algorithm. In section 7.3 the self-learning sexual genetic algorithm will be compared with SMAC [53], which is a more recent tuning algorithm to replace ParamILS. Section 7.4 is the conclusion of this chapter.

7.1 Introduction

The self-learning sexual genetic algorithm is a hybrid algorithm based on the sexual genetic algorithm and the self-learning genetic algorithm. The self-learning sexual genetic

algorithm (SLSGA) pseudocode (Algorithm 5) introduces SLSGA's working principle and shows how self-learning combines with the sexual genetic algorithm. Compared with the self-learning genetic algorithm, the self-learning sexual genetic algorithm spends time on fitness evaluation and selects k elitisms from the male group, instead of from the whole population. The aim of the self-learning sexual genetic algorithm is the same as self-learning genetic algorithm that finds a best average performance parameter set from the training sets. Meanwhile, a penalty strategy will be implemented in the self-learning sexual genetic algorithm.

The flowchart of the self-learning sexual genetic algorithm clearly demonstrates how it works from the population initialisation to optimisation completion. The following is the way to implement it:

Initialization - As in self-learning genetic algorithm, a few populations were firstly initialised such as the starting population P for the evolutionary and the best population PB. At begin the mating population is initialized as the starting population P, else it is the total of the population P and the best population PB. According to the value of the tuning parameters, the chromosome length $choromlength$ and the best parameter set $BestParam$ will be initialized.

Bonus and Penalty Strategy - As the bonus strategy in the self-learning genetic algorithm, a penalty strategy is initialised to collect statistic of the worst parameter set in each generation. Therefore array $PenaltyRecorder_k$ is initialised to record the occurrence frequency of the worst or invalid parameter sets that happened in each generation, where k is the size of array $PenaltyRecorder_k$. As in the bonus strategy, the recent best parameter set will replace the worst or the early on in $PenaltyRecorder_k$, when k is less than the generation size.

$PenaltyRecorder_k$ is used to record the worst or the last invalid parameter set happened in each generation. Array B is to record the occurrence frequency of each best parameter that happened in each generation as in self-learning genetic algorithm.

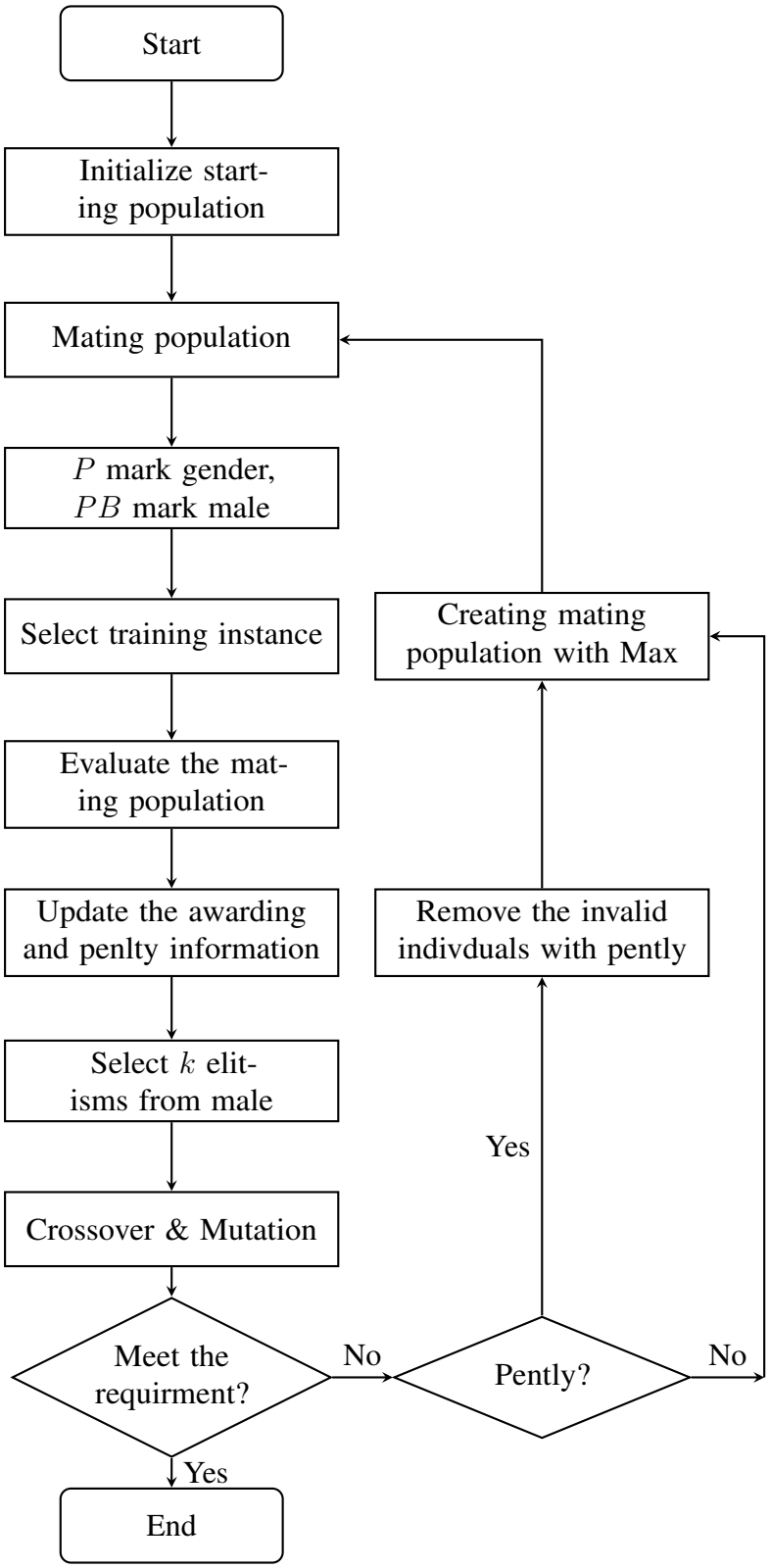


Figure 7.1: The Flowchart of Sexual Self-learning Genetic Algorithm

Algorithm 5 Self-learning Sexual Genetic Algorithm

Initialize $P, PB, B, Male, Female, PenaltyRecorder$
for $i \leftarrow 1$ to n **do**
    $T \leftarrow ChooseInstance(TrainingSets)$
    $PC \leftarrow RandomMarkGender(P)$;
    $Female \leftarrow P \setminus PC$;
    $Male \leftarrow PC \cup PB$;
    $BestParam\&Invalids(orWorst) \leftarrow$ Fitness($Male$)
    for $k \leftarrow 1$ to $i - 1$ **do**
        if $BestParam == B_k$ **then**
            $B_k \leftarrow B_k + 1$
        else
            $B_i \leftarrow$ best configuration
        end if
    end for
    if $Invalids(orWorst) == PenaltyRecorder_k$ **then**
        $PenaltyRecorder_k = PenaltyRecorder_k + 1$
    else
        $PenaltyRecorder_i \leftarrow Invalids(orWorst)$
    end if
    $Parent1 \leftarrow Select(Male)$
    $Parent2 \leftarrow Select(Female)$
    for $j \leftarrow 1$ to n **do**
        $P \leftarrow$ Crossover(Mutation($Parents1$),Mutation($Parents2$))
    end for
    if Checktime()==true **then**
        break
    else if Penalty()==true **then**
        replace()
    end if
end for
if Bouns()==true **then**
    $\lambda \leftarrow B_{best}$
else
    $\lambda \leftarrow BestParam$
end if
return λ

Another task of the penalty strategy is to remove the possible bad parameter value from the parameter sets. To realise it, a trigger value $PenaltyTrigger$ was set to decide the time to remove the parameter. Therefore a trigger function Penalty(), which checks whether the occurrence times of the worst parameter occurred most time is larger than the trigger value $PenaltyTrigger$, was created and implemented.

Once the penalty strategy is triggered, the replace() function will firstly find the parameter values which are not in any parameter sets in BP but in the worst parameter set. Those parameter values will be prevented and replaced by random values with the possibility α in the new generation. The value of the α is assigned by the occurrence frequency of the worst parameter set. For example if the worst parameter set appears five times in ten generations, α is 0.5 (5/10). When the self-learning genetic algorithm finished, the bonus mechanism will find out an optimal parameter set which has highest occurrence frequency or is the best parameter in the last generation.

Mating Rule - As in the self-learning genetic algorithm, a random instance T will be chosen from the training sets for each generation. At the beginning of each generation, the population P randomly marks half the population as male by $RandomMarkGender()$ and store the male in the population PC. The rest of parameter in P were marked as $Female$ population. The parameter sets in the best array B are all marked as male. The parameter sets in the population $Male$ will be evaluated to gain the best and worst/invalid parameter set which will be stored for bonus and penalty strategy. m elitisms will be selected from the male population as one mating parent. Each individual in $Female$ population has the same possibility to mating as another mating parent.

Crossover and Mutation - As in the self-learning genetic algorithm, the single point crossover will be applied in this chapter. The self-learning genetic algorithm will choose the most classic one point mutation.

Train Instance	Test Instance	SLGA (s)	SLSGA(s)
26-Queens 27-Queens 28-Queens	100-Queens	0.2s	0.11s
BIBD1 BIBD2 BIBD3	BIBD (9,90,40,4,15)	2911s	1381s
Golomb (6) Golomb (7) Golomb (8)	Golomb(11)	4169s	3484s
Langford(2,10) Langford(3,17) Langford(3,19)	Langford(2,27)	1223s	1079s

Table 7.1: The Performance Comparison in Tuning Minion

7.2 Self-learning Sexual Genetic Algorithm vs. Self-learning Genetic Algorithm in Tuning Minion

The self-learning sexual genetic algorithm was proposed by the instance-based tuning algorithm self-learning genetic algorithm. Compared with self-learning genetic algorithm, it combined the idea of the sexual genetic algorithm. It means that only half of the population was evaluated their parameter sets. Although the evaluation time is saved, the change to evaluate more parameter sets lost. To extract more information in the evaluation of half population and improve its efficiency, the penalty strategy was discussed in the self-learning sexual genetic algorithm.

To justify its performance, the self-learning genetic algorithm will compare with the self-learning genetic algorithm in tuning Minion. In the comparison there will be three training instances and one testing instance for each CSPs as in last chapter. The population size will be 10, crossover rate is 0.9 and mutation rate is 0.1. The amount of tuning time is assigned 300 seconds for both algorithms. The number of parallel tasks for both is 8. The k in the bonus strategy and the penalty strategy is the same as population size. To clearly demonstrate the performance of the penalty strategy, $PenaltyTrigger$ is assigned 2. Each tuning trial was run 10 times and we observed the average of the minimums.

Table 7.1 shows the average of the solving time for the large test instance with the found optimal parameter sets. The left columns introduces the name of the training instances and the test instance. The data in the right columns clearly indicates the self-learning sexual genetic algorithm outperforms its precursor in tuning four CSPs. It saves running cost on the fitness evaluation in each generation since it just need to evaluate the fitness of the half male population. In another word, it means it has more generations in the same amount of tuning time.

7.3 Self-learning Sexual Genetic Algorithm vs. SMAC in Tuning SAPS

To verify the performance of self-learning genetic algorithm, it will be compared with the Sequential Model-based Algorithm Configuration (SMAC) which is a successor of ParamILS. They will be implemented in tuning SAPS to compare their efficiency. The same benchmark set as in the last chapter, which consists of 113 SAT instances for training and 100 different instances for testing, was chosen for SAPS. The self-learning genetic algorithm runs the tuning for 200 generations with a population of size 60 and a $PenaltyTrigger$ of value 2. The crossover rate will be 0.9 and mutation rate will be 0.1. The total CPU time for tuning is 1800 seconds which was used for self-leaning genetic algorithm and SMAC when configuring SAPS. The cutoff time for evaluating each parameter set in each training instance for both tuning approaches is one second.

Table 7.2 shows the average performance of both algorithms in tuning SAPS on 20 runs. It clearly indicates that self-learning sexual genetic algorithm outperform SMAC in tuning SAPS. The self-learning genetic algorithm's worst parameter set results in a performance of 9.5 ms per training instance, which is better than the mean of SMAC parameter set which requires 9.98 ms. Meanwhile the worst parameter set of self-learning genetic algorithm in testing instance is 8.9 which is still little better than 8.91 that is the average performance of SMAC. Although the performance of self-learning sexual genetic

	SMAC		SLSGA	
Run	Training(ms)	Testing(ms)	Training(ms)	Testing(ms)
1	11.4	8	8.2	7.6
2	9.8	8.8	7.0	7.2
3	10.1	9.6	8.9	8.5
4	9.7	8.4	7.1	7.7
5	9.6	10.4	7.3	7.5
6	10.2	10.3	6.6	8.8
7	10.6	9.0	7.2	6.8
8	9.9	10.4	7.5	7.4
9	9.7	8.0	6.8	7.5
10	10.5	11.2	8.3	8.9
11	9.9	10.8	6.8	7.6
12	10.3	8.2	7.5	6.4
13	10.2	8.6	7.1	8.3
14	9.1	9.0	8.1	5.2
15	9.9	8.8	9.5	8.5
16	10.4	6.9	7.4	7.9
17	10.1	8.2	6.5	8.4
18	9.6	8.6	7.8	9.2
19	9.1	6.4	7.7	6.8
20	9.4	8.7	7.9	7.7
Mean	9.98	8.91	7.56	7.70
STDDEV	0.52	1.21	0.75	0.92

Table 7.2: The performance of SLSGA and SMAC in tuning SAPS on 20 runs

algorithm in the testing instance is not better than in the training instance, it still shows its efficiency and superiority. The standard deviation shows that the self-learning sexual genetic algorithm is more stable than SMAC no matter in training instance or testing instances.

7.4 Conclusions

As the second instance-based tuning algorithm in this thesis, the self-learning sexual genetic algorithm is proposed by combining the idea of sexual genetic algorithm and self-learning genetic algorithm.

The main aim of self-learning sexual genetic algorithm is to extract more useful information for finding an optimal parameter set with less running time. Because of the idea of the sexual genetic algorithm, the self-learning sexual genetic algorithm reduces half fitness evaluation time. The self-learning sexual genetic algorithm attempts to extract more useful information by bonus and penalty strategy.

The experiment result shows the application of the bonus and penalty strategy successfully helped the algorithm to find an optimal parameter set with less time on the evaluation by combining the male and female (competitive and co-operative) strategy. Although self-learning sexual genetic algorithm is efficient on tuning, the size choice of $PenaltyTrigger$ and the possibility α in penalty strategy is worth further study.

Chapter 8

Conclusion

8.1 Summary

This thesis firstly analysed the performance of the operators and their features in standard genetic algorithm. It embodied the powerful search ability of the genetic algorithm and its parameter sensitivity. The experiment result indicated that GAs could approach the best fitness in a few generations. It also showed that the lower mutation rate and high crossover is a better choice for the experiment in this thesis.

In this thesis, genetic-based algorithms were chosen to help solvers on single instance tuning and instance-based tuning. There are two main reasons to choose GAs to implement solver tuning:

- GAs have a powerful ability to tackle optimisation problems which lack auxiliary information
- GAs perform parallel search rather than linear search; each chromosome (solution to the problem) competes against others in each generation

For the single instance tuning, two algorithms GACM and a sexual genetic algorithm were proposed in this thesis. GACM demonstrated the feasibility that could bridge the genetic algorithms and the constraint solvers.

To improve the tuning efficiency in further, a sexual genetic algorithm which implements the elitism and parallel mechanism was proposed to deal with the singe instance tuning. It has demonstrated that the elitisms strategy in the SGA is very important and that selecting suitable elitisms percentages leads to an ideal optimisation speed. The sexual genetic algorithm is more efficient than standard GAs in preprocessing selection; it is not necessary for the sexual genetic algorithm to evaluate the fitness of all chromosomes, which is a considerable consumer of CPU time.

The self-learning genetic algorithm is an instance-based tuning algorithm which acquires, and discovers new knowledge form the training set. Beside the learning ability of genetic algorithm itself, a statistical iteration strategy was implemented in the self-learning genetic algorithm to extract the optimal parameter set from training set. This statistical iteration strategy also called bonus strategy could help the sexual genetic algorithm keep a record of all best parameter set in each generation. The experiments show that the sexual genetic algorithm is efficient for tuning solver by comparing with ParamILS.

Another new instance-based algorithm called self-learning sexual genetic algorithm was proposed in this thesis. It was created by combining the idea of sexual-genetic algorithm and self-learning genetic algorithm. To achieve a better desired result, a penalty was be applied in the self-learning genetic algorithm.

8.2 Contributions

According to the three main research questions, there are three key academic contributions from the thesis.

Firstly, this thesis developed an idea of the framework that combined the genetic algorithms with the solver Minion and SAPS. The experiment results showed that the framework is feasible and acceptable. The framework provides a practical way which could tune the Minion properly. It also showed the search ability of the genetic algorithm in the autonomous search.

Secondly, this thesis explored the parameter sensitivity of the genetic algorithms in different situations. Goldberg pointed out that the parameter setting in the genetic algorithm itself is hard to control. The experiments in this thesis illustrated the parameter sensitivity of the genetic algorithms in various situations. Those experiment results could be used for future work as the benchmark.

Finally, this thesis proposed a few genetic algorithms for tuning such as the self-learning sexual genetic algorithm, which is based on the sexual genetic algorithm and the self-learning genetic algorithm. Those genetic based algorithms give a feasible and efficient solution(s) for solvers tuning.

8.3 Future Work

The results showed that four algorithms discussed in the thesis are successful in tuning different types of instances and solver. However, a number of challenges remain for future exploration. Although four genetic-based tuning algorithms showed their performance in tuning Minion and SAPS by comparing with other existing approaches such as GGA and SMAC, more different solvers were expected to choose to justify their robustness. In the self-learning genetic algorithm, the size of the best array in the bonus strategy is worthwhile to explore in further. The self-learning sexual genetic algorithm implements a new penalty strategy to improve the performance. The effect of the $PenaltyTrigger$ and the replace possibility α in the penalty strategy is another possible direction to investigate.

Bibliography

[1] Omar Arif Abdul-Rahman, Masaharu Munetomo, and Kiyoshi Akama. An adaptive parameter binary-real coded genetic algorithm for constraint optimization problems: Performance analysis and estimation of optimal control parameters. *Information Sciences*, 233:54 – 86, 2013.

[2] Belarmino Adenso-Daz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, 54(1):99–114, 2006.

[3] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5):443–462, 2002.

[4] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2004.

[5] C. Ansótegui, M. Sellmann, and K. Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In *Principles and Practice of Constraint Programming-CP 2009: 15th International Conference, CP 2009 Lisbon, Portugal, September20-24, 2009 Proceedings*, page 142. Springer, 2009.

[6] David L Applegate. *The traveling salesman problem: a computational study*. Princeton University Press, 2006.

[7] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221:179 – 210, 1999.

[8] C. Audet and J. Dennis. Analysis of generalized pattern searches. *SIAM Journal on Optimization*, 13(3):889–903, 2002.

[9] Charles Audet and John E Dennis Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on optimization*, 17(1):188–217, 2006.

[10] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[11] T. Back. Selective pressure in evolutionary algorithms: a characterization of selection mechanisms. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 57–62 vol.1, Jun 1994.

[12] T. Bartz-Beielstein, C.W.G. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pages 773–780 Vol.1, Sept 2005.

[13] J. Christopher Beck, Patrick Prosser, and Richard J. Wallace. Toward understanding variable ordering heuristics for constraint satisfaction problems. In *IN: PROCEEDINGS OF THE FOURTEENTH IRISH ARTIFICIAL INTELLIGENCE AND COGNITIVE SCIENCE CONFERENCE*, pages 11–16, 2003.

[14] J. Christopher Beck, Patrick Prosser, and Richard J. Wallace. Trying again to fail-first. In *In: Recent Advances in Constraints. Papers from the 2004 ERCIM/CologNet Workshop-CSCLP 2004. LNAI No. 3419*, pages 41–55. Springer, 2005.

[15] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in chip. *Mathematical and computer Modelling*, 20(12):97–123, 1994.

[16] Alberto Bertoni and Marco Dorigo. Implicit parallelism in genetic algorithms, 1993.

[17] Christian Bessiere. Constraint propagation. *Handbook of constraint programming*, pages 29–83, 2006.

[18] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Sttzle. F-race and iterated f-race: An overview, 2009.

[19] Avrim L Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1):245–271, 1997.

[20] Erick Cant-Paz. A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES, RESEAUX ET SYSTEMS REPARTIS*, 10, 1998.

[21] Alain Colmerauer. An introduction to prolog iii. In *Computational Logic*, pages 37–79. Springer, 1990.

[22] Charles Darwin. On the origins of species by means of natural selection. *London: Murray*, 1859.

[23] Charles Darwin and William F Bynum. *The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life*. AL Burt, 2009.

[24] Lawrence Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'85, pages 162–164, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[25] Kenneth Alan De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, Ann Arbor, MI, USA, 1975. AAI7609381.

[26] Kalyanmoy Deb et al. *Multi-objective optimization using evolutionary algorithms*, volume 2012. John Wiley & Sons Chichester, 2001.

[27] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A clause-based heuristic for sat solvers. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2005.

[28] M. Dib, R. Abdallah, and A. Caminada. Arc-consistency in constraint satisfaction problems: A survey. In *Computational Intelligence, Modelling and Simulation (CIMSiM), 2010 Second International Conference on*, pages 291–296, Sept 2010.

[29] Ulrich Dorndorf, Erwin Pesch, and Ton Phan-Huy. Constraint propagation techniques for the disjunctive scheduling problem. *Artificial Intelligence*, 122(1?):189 – 240, 2000.

[30] Amer Draa, Souham Meshoul, Hichem Talbi, and Mohamed Batouche. A quantum-inspired differential evolution algorithm for solving the n-queens problem. *International Arab Journal of Information Technology (IAJIT)*, 7(1), 2010.

[31] H.A. Eiselt and C.-L. Sandblom. Introduction to operations research. In *Operations Research*, pages 1–12. Springer Berlin Heidelberg, 2010.

[32] Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The adaptive constraint engine. In *In: CP 02 Principles and Practice of Constraint Programming*, pages 525–540. Springer, 2002.

[33] Larry J. Eshelman and David J. Schaffer. Preventing premature convergence in genetic algorithms by preventing incest. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122. San Francisco, CA: Morgan Kaufmann, 1991.

[34] Eugene C Freuder and Mark Wallace. Constraint programming. In *Search Methodologies*, pages 369–401. Springer, 2014.

[35] Alan M Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In *International Joint Conference on Artificial Intelligence*, volume 19, page 109. LAWRENCE ERLBAUM ASSOCIATES LTD, 2005.

[36] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *Proceedings of the 17th Eureopean Conference on Artificial Intelligence (ECAI'06)*, pages 98–102, 2006.

[37] Ian P Gent, Ian Miguel, and Andrea Rendl. Tailoring solver-independent constraint models: A case study with essence and minion. In *Abstraction, Reformulation, and Approximation*, pages 184–199. Springer, 2007.

[38] Ian P Gent and Toby Walsh. Csplib: a benchmark library for constraints. In *Principles and Practice of Constraint Programming–CP99*, pages 480–481. Springer, 1999.

[39] Kai Song Goh, Andrew Lim, and Brian Rodrigues. Sexual selection for genetic algorithms. *Artificial Intelligence Review*, 19(2):123–152, 2003.

[40] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[41] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Urbana*, 51:61801–2996, 1991.

[42] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers, 2008.

[43] Jonathan M Gratch. Composer: A decision-theoretic approach to adaptive problem solving. Technical report, Champaign, IL, USA, 1993.

[44] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum, New Jersey (160-168), 1985.

[45] John J Grefenstette. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Psychology Press, 2013.

[46] Jun Gu. Efficient local search for very large-scale satisfiability problems. *ACM SIGART Bulletin*, 3(1):8–12, 1992.

[47] Zineb Habbas, Michaël Krajecki, and Daniel Singer. The langford's problem: A challenge for parallel resolution of csp. In *Proceedings of the th International Conference on Parallel Processing and Applied Mathematics-Revised Papers*, PPAM '01, pages 789–796, London, UK, UK, 2002. Springer-Verlag.

[48] Youssef Hamadi, Eric Monfroy, and Frdric Saubion. An introduction to autonomous search. In Youssef Hamadi, Eric Monfroy, and Frdric Saubion, editors, *Autonomous Search*, pages 1–11. Springer Berlin Heidelberg, 2012.

[49] Nikolaus Hansen. The cma evolution strategy: A comparing review. In JoseA. Lozano, Pedro Larraaga, Iaki Inza, and Endika Bengoetxea, editors, *Towards a New Evolutionary Computation*, volume 192 of *Studies in Fuzziness and Soft Computing*, pages 75–102. Springer Berlin Heidelberg, 2006.

[50] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263 – 313, 1980.

[51] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. 1992.

[52] Sophie Huczynska, Paul McKay, Ian Miguel, and Peter Nightingale. Modelling equidistant frequency permutation arrays: An application of constraints to mathematics. In IanP. Gent, editor, *Principles and Practice of Constraint Programming - CP*

2009, volume 5732 of *Lecture Notes in Computer Science*, pages 50–64. Springer Berlin Heidelberg, 2009.

[53] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Parallel algorithm configuration. In *Proc. of LION-6*, pages 55–70, 2012.

[54] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[55] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Int. Res.*, 36(1):267–306, September 2009.

[56] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, AAAI'07, pages 1152–1157. AAAI Press, 2007.

[57] Ian Miguel Ian P. Gent, Chris Jefferson. Watched literals for constraint propagation in minion. *In Proc. CP2006, 182197*, pages 182–197, 2006.

[58] Joxan Jaffar and J-L Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

[59] Joxan Jaffar, Spiro Michaylov, Peter J Stuckey, and Roland HC Yap. The clp (r) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, 1992.

[60] Dietmar Jannach and Markus Zanker. Modeling and solving distributed configuration problems: A csp-based approach. *Knowledge and Data Engineering, IEEE Transactions on*, 25(3):603–618, 2013.

[61] Bart MP Jansen and Stefan Kratsch. Data reduction for graph coloring problems. *Information and Computation*, 231:70–88, 2013.

[62] Tommy R Jensen and Bjarne Toft. *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.

[63] Yuejun Jiang, Henry Kautz, and Bart Selman. Solving problems with hard and soft constraints using a stochastic algorithm for max-sat, 1995.

[64] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac-instance-specific algorithm configuration. *ECAI*, 215:751–756, 2010.

[65] Leveen Kanal and Vipin Kumar. *Search in artificial intelligence*. Springer Publishing Company, Incorporated, 2012.

[66] Lars Kotthoff, Ian Miguel, and Peter Nightingale. Ensemble classification for constraint solver configuration. In *Principles and Practice of Constraint Programming–CP 2010*, pages 321–329. Springer, 2010.

[67] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[68] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32, 1992.

[69] Tony Lambert, Carlos Castro, Eric Monfroy, Mara Riff, and Frdric Saubion. *Hybridization of Genetic Algorithms and Constraint Propagation for the BACP*, volume 3668 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005.

[70] Jean-Louis Lauriére. *Problem Solving and Artificial Intelligence*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[71] Stuart P Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[72] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter Van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, volume 3, pages 245–250, 2003.

[73] Sushil J Louis and Judy Johnson. Solving similar problems using genetic algorithms and case-based memory. In *ICGA*, pages 283–290. Citeseer, 1997.

[74] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[75] Kim Marriott and Peter J Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[76] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[77] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[78] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences*, 7:95–132, 1974.

[79] Heinz Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In *PPSN*, volume 92, pages 15–25, 1992.

[80] Michael Negnevitsky. *Artificial intelligence: a guide to intelligen systems*. Pearson Education, 2005.

[81] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in savile row through associative-commutative common subexpression elimination. In *Principles and Practice of Constraint Programming*, pages 590–605. Springer, 2014.

[82] Eoin OMahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry OSullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[83] Marc Pomplun, Tyler W Garaas, and Marisa Carrasco. The effects of task difficulty on visual search strategy in virtual 3d displays. *Journal of vision*, 13(3), 2013.

[84] David Poole and Alan K. Mackworth. Artificial intelligence - foundations of computational agents. 2010.

[85] Steven Prestwich. A local search algorithm for balanced incomplete block designs. In *Recent Advances in Constraints*, pages 132–143. Springer, 2003.

[86] MM Raghuwanshi and OG Kakde. Genetic algorithm with species and sexual selection. In *Cybernetics and Intelligent Systems, 2006 IEEE Conference on*, pages 1–8. IEEE, 2006.

[87] M.M. Raghuwanshi and O.G. Kakde. Genetic algorithm with species and sexual selection. In *Cybernetics and Intelligent Systems, 2006 IEEE Conference on*, pages 1–8, 2006.

[88] Jason Reed. Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 49–56, New York, NY, USA, 2009. ACM.

[89] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[90] John R Rice. The algorithm selection problem. 1975.

[91] Simon Rogers and Mark Girolami. *A First Course in Machine Learning*. Chapman & Hall/CRC, 1st edition, 2011.

[92] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[93] Ranjit K Roy. *A primer on the Taguchi method*. Society of Manufacturing Engineers, 2010.

[94] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[95] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 44(1.2):206–226, Jan 2000.

[96] Jose Sanchez-Velazco and John A. Bullinaria. Sexual selection with competitive/co-operative operators for genetic algorithms. In *In IASTED International Conference on Neural Networks and Computational Intelligence (NCI*. ACTA Press, 2003.

[97] Douglas C Schmidt and Larry E Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM (JACM)*, 23(3):433–445, 1976.

[98] Adam Słowik. Steering of balance between exploration and exploitation properties of evolutionary algorithms-mix selection. In *Artifical Intelligence and Soft Computing*, pages 213–220. Springer, 2010.

[99] Barbara M. Smith. A tutorial on constraint programming. Technical report, University of Leeds, 1995.

[100] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *In: Thirteenth European Conference on Artificial Intelligence (ECAI 98*, pages 249–253. John Wiley and Sons, 1997.

[101] Barbara M. Smith and Ian P.Gent. *Constraint Modelling Challenge*. 2005.

[102] William M Spears and Vic Anand. *A study of crossover operators in genetic programming*. Springer, 1991.

[103] William M Spears and Kenneth A De Jong. An analysis of multi-point crossover. Technical report, DTIC Document, 1990.

[104] Timothy Swanson. *Global action for biodiversity: an international framework for implementing the convention on biological diversity*. Routledge, 2013.

[105] Andrea Toffolo and Ernesto Benini. Genetic diversity as an objective in multi-objective evolutionary algorithms. *Evolutionary Computation*, 11(2):151–167, 2003.

[106] Edward Tsang. *Foundations of Constraint Satisfaction*. 1993.

[107] M. Jalali Varnamkhasti and MasoumehVali. Sexual selection and evolution of male and female choice in genetic algorithm. *Scientific Research and Essays*, 7(31):2788–2804, 2012.

[108] Stefan Wagner and Michael Affenzeller. Sexualga: Gender-specific selection for genetic algorithms. In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI)*, volume 4, pages 76–81. Citeseer, 2005.

[109] Qiu Weisheng. N queens problem. *Journal of Mathematics*, 2:002, 1986.

[110] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[111] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, 43(14):817 – 831, 2001.

[112] L Darrell Whitley et al. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *ICGA*, volume 89, pages 116–123, 1989.

[113] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.